THESIS ON INFORMATICS AND SYSTEM ENGINEERING C116

Certification of Context-Free Grammar Algorithms

DENIS FIRSOV

TUT
PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Institute of Cybernetics

This dissertation was accepted for the defense of the degree of Doctor of Philosophy in Informatics on 26 July 2016.

Supervisor: Prof. Tarmo Uustalu, PhD
Institute of Cybernetics
Tallinn University of Technology
Tallinn, Estonia

Opponents: Prof. Gert Smolka, Dr. rer. nat.
Fachrichtung Informatik
Universität des Saarlandes
Saarbrücken, Germany

Simão Melo de Sousa, PhD
Departamento de Informática
Universidade da Beira Interior
Covilhã, Portugal

Defense: 31 August 2016

Declaration: Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been previously submitted for any degree or examination.

/Denis Firsov/

ISSN 1406-4731
ISBN 978-9949-83-005-3 (publication)
ISBN 978-9949-83-006-0 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C116

Kontekstivabade grammatikate algoritmide sertifitseerimine

DENIS FIRSOV

TTÜ
KIRJASTUS

Contents

List of publications

The thesis is an overview of and is based on publications I–III below. Publications IV and V are on related subjects.

In the thesis, the publications are referred to as Papers I–V. All publications have undergone a rigorous peer review process.

I D. Firsov, T. Uustalu. Dependently typed programming with finite sets. In *Proc. of 2015 ACM SIGPLAN Wksh. on Generic Programming, WGP '15 (Vancouver, BC, Aug. 2015)*, pp. 33–44. ACM Press, 2015.

II D. Firsov, T. Uustalu. Certified CYK parsing of context-free languages. *J. of Log. and Algebr. Meth. in Program.*, v. 83(5–6), pp. 459–468, 2014.

III D. Firsov, T. Uustalu. Certified normalization of context-free grammars. In *Proc. of 4th ACM SIGPLAN Conf. on Certified Programs and Proofs, CPP '15 (Mumbai, Jan. 2015)*, pp. 167–174. ACM Press, 2015.

IV D. Firsov, T. Uustalu. Certified parsing of regular languages. In G. Gonthier, M. Norrish, eds., *Proc. of 3rd Int. Conf. on Certified Programs and Proofs, CPP 2013 (Melbourne, Dec. 2013)*, v. 8307 of *Lect. Notes in Comput. Sci.*, pp. 98–113. Springer, 2013.

V D. Firsov, T. Uustalu, N. Veltri. Variations on Noetherianness. In R. Atkey, N. Krishnaswamy, eds., *Proc. of 6th Wksh. on Mathematically Structured Functional Programming, MSFP 2016 (Eindhoven, 2016)*, *Electron. Proc. in Theor. Comput. Sci.*, pp. 76–88. Open Publishing Assoc., 2016.

Author's contribution to the publications

The candidate is responsible for designing and implementing the algorithms and proofs in Agda both for the publications the thesis is based on well as the thesis itself. In addition, the candidate had the lead role in conceiving, drafting and producing the manuscripts. Also the work was presented at the conferences by the thesis author.

The general idea of implementing a certified parser generator for context-free grammars was conceptualized and motivated by the candidate's supervisor. The supervisor also helped in the writing.

The lead ideas for the work on finite sets (Paper I) are the candidate's.

Accompanying code

Papers I–III are accompanied with developments (programs, proofs) in the Agda dependently typed programming language [1, 38]. The code associated with the papers is located at:

- `http://cs.ioc.ee/~denis/finset` (Paper I).
- `http://cs.ioc.ee/~denis/cert-cfg` (Paper II).
- `http://cs.ioc.ee/~denis/cert-norm` (Paper III).

The Agda code of the thesis is located at `http://cs.ioc.ee/~denis/phd`. It is a streamlined adaptation of the above-mentioned developments.

The differences come mostly from the changes introduced in newer versions of Agda and its standard library. The current development uses the latest releases of Agda and Agda Standard Library (for the moment these are Agda 2.5.1 and Std. Lib. 0.12).

We also made the interfaces of earlier developments compatible with each other.

Introduction

Motivation

As the recognition of information technology broadens to new fields, the need for reliable and fast development of new programming languages has become more important than ever. Nowadays, building languages targeted at a particular problem domain is considered a standard technique in software engineering [18].

However, programming languages are software systems as well. They are usually defined by a compiler or an interpreter. The importance of the correctness of the implementation of the compiler is very high, since this property is propagated to the software that is written in that language—if the compiler has defects, then programs compiled with it also perform poorly or are flawed. The correctness of a compiler means that the compiled code should behave in the "same" way as the source program—according to the semantics assigned to the source and target languages [2].

The correctness of a compiler (in the above mentioned sense) for a higher-level programming language is thus of crucial importance. The compilation process is divided into a number of phases. Parsing is the phase of structuring the input code into a tree in conformance with the grammar of the programming language. The later stages of the compilation process depend on the correctness of the parser used. Therefore, it becomes important to investigate the parsing techniques in a formal setting.

Problem statement

The main goal of the thesis is to implement a set of tools and a certified parsing algorithm for general context-free grammars, using the Agda dependently typed programming language [1, 38]. More precisely, the main interest is to implement a function that, given a context-free grammar and a string, finds a derivation (parse tree) of the string in the grammar provided. Moreover, the correctness proof must ensure that only valid derivations can be delivered, and that derivation, or all derivations, will be delivered, if one exists.

Contribution of the thesis

The main contributions of the thesis towards the implementation of a certified parser generator are:

- An investigation of the relationships between different encodings of listable sets and their implication on decidable equality.
- A toolbox for boilerplate-free programming with finite sets given by listing a subset of some base set with decidable equality.

- A certified implementation of the CYK parsing algorithm for context-free grammars in Chomsky normal form.
- A certified implementation of conversion of general context-free grammars to Chomsky normal form. The proofs of soundness and completeness of the conversion are functions for conversion between parse trees for the normalized and original grammars.
- Obtained by combining the above, a certified implementation of a parsing algorithm for general context-free grammars.

Outline of the thesis

Section 1 provides the basic background for this thesis. It gives an overview of the compilation process and possible approaches to formal verification. We start by outlining the usual phases of a compiler and describe the parsing phase in more detail. Then we discuss different approaches to formal verification of software systems. Finally, we explain our decision to work in a constructive setting by describing its main properties. Also we give a short introduction to the Agda language.

As the problem under investigation requires us to work with finite sets, we devote Section 2 (corresponding to Paper I) to exploring different notions of finite sets in the constructive setting. We start by giving different encodings of listability of a set and proving that the encodings are logically equivalent. Then we prove that listable sets have decidable equality. Finally, we develop a toolbox for boilerplate-free programming with finite sets given by listing a subset of some base set with decidable equality.

In Section 3, (summarizing the Paper II) we implement the Cocke–Younger–Kasami (CYK) algorithm of parsing context-free languages and prove the correctness of the implementation. We start by giving a simple recursive version of the algorithm. The simplicity of this naive implementation allows us to prove the correctness concisely. But it suffers from excessive recomputations. Therefore, we refine the initial implementation into the efficient memoizing algorithm. We then prove that the memoizing algorithm is equivalent to the first one.

The CYK algorithm requires context-free grammars in Chomsky normal form. Therefore, in Section 4 (for Paper III), we develop a normalization function that turns a general context-free grammar into a normalized one. Next, we prove that the implemented transformation is sound and complete. The soundness and completeness proofs are functions for conversion between parse trees for the normalized and original grammars. Finally, we combine the results of Papers II and III into a certified parser generator for general context-free grammars.

Section 5 gives an overview of the related work. Section 6 concludes the thesis and describes possible directions for future work.

1 Background

In this section, we outline the basic background for our work. First, we list the phases of the typical process of compilation and describe in particular the phase of syntax analysis. Next, we discuss different approaches to formal verification and motivate our decision to work in the constructive and dependently typed setting of the Agda programming language. Then we explain the main ideas of constructive logic, the Curry–Howard correspondence and provide a short introduction into the basics of programming in Agda.

1.1 Compilers and parsing

Compilation is a process of translating a program written in a high-level language into a machine language. It is usually divided into a number of phases [3]:

1. lexical analysis,
2. syntax analysis,
3. semantic analysis,
4. optimisation,
5. code generation.

This subdivision into phases is helpful for coping with the complexity of compilers. There is little overlap between the phases and each phase has a concise interface toward the neighbouring phases [2]. In this work, we are mostly interested in syntax analysis.

Traditionally, the lexical and syntactic analyses of a programming language are described by using formal grammars. The lexical aspect is usually a regular language, specified by a regular expression. A regular expression defines the set of possible character sequences that are used to form individual tokens or lexemes. The syntactic analysis is done with reference to a context-free grammar which recursively defines the components that can make up an expression out of tokens [26]. More formally, a context-free grammar is a 4-tuple $G = (N, T, R, S)$, where:

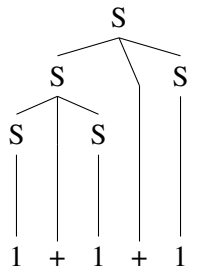
1. N is a finite set; each element $A \in N$ is called a nonterminal.
2. T is a finite set of terminals, disjoint from N. The set of terminals is the alphabet of the language defined by the grammar G. In a compiler, the terminals of syntax analysis are the lexical tokens.
3. R is a finite set of production rules. A rule is usually denoted by an arrow notation as $A \longrightarrow \gamma$, where $A \in N$ and γ is a sequence of nonterminals and terminals.

4. S is the start nonterminal form the set N.

A context-free grammar G is said to be in *Chomsky normal form* if all of its production rules are either of the form $A \longrightarrow BC$ or $A \longrightarrow t$, where A, B, C are nonterminals and t is a terminal; B and C cannot be start nonterminal. Additionally, there must be a flag which indicates if the empty word is in the language of G [46]. Perhaps surprisingly at first, every context-free language can be specified by a grammar in Chomsky normal form.

Next, we discuss derivations in some fixed context-free grammar G. Let $\alpha A \beta$ be some sequence of symbols (terminals and nonterminals), and A be a nonterminal. If there is a rule $A \longrightarrow \gamma$ in R then we can *derive* $\alpha\gamma\beta$ from $\alpha A \beta$ (denoted by $\alpha A \beta \Rightarrow \alpha\gamma\beta$). Let $\overset{+}{\Rightarrow}$ be the transitive closure of the derivation relation $\Rightarrow$. Then the *language* of the grammar G is the set of all strings (sequences of terminals) derivable from the nonterminal S in the sense of the relation $\overset{+}{\Rightarrow}$.

A derivation tree or parse tree is an ordered rooted tree that represents the syntactic structure of the derived string. Rendering a derivation as a tree helps one to grasp the structure of the parsed string. For example, let us define a grammar with a single nonterminal S and terminals 1 and $+$. There are two rules $S \longrightarrow 1$ and $S \longrightarrow S + S$. Then the following is a possible derivation tree of the string "1+1+1":



After specifying the formal grammar, a *parser generator* is able to produce an executable *parser*. A parser for a regular language (also called a *lexer*) breaks the source code text into tokens. A parser for a context-free language identifies the syntactic structure of the program by building a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of the grammar. The parse tree is then analyzed and transformed by the subsequent phases of the compiler [3].

The correctness of a parser generator is of significant importance, since the overall correctness of a compiler depends on the correctness of each of its phases, in particular syntax analysis. The parser generator is correct, if, for any suitable input grammar, it generates a program that parses all strings in the language of the grammar and for others flags an error.

1.2 Formal verification

The correctness of a software system is usually established by full formal verification. The specification of the system is presented in some mathematical language. The system itself also must be modeled in some mathematical formalism. The verification is then done by providing a formal proof that the model satisfies the specification. It is important to ensure that the mathematical model of the system describes all the aspects relevant to the correctness of the actual system.

Model checking is an approach to formal verification [34]. In model checking, the model usually consists of states and transitions between them. The properties to be verified can be given in temporal logics (e.g., linear temporal logic). The verification of such systems can be done by exhaustively exploring all traces. The advantage of model checking is that it is automatic, but the downside is that it does not easily scale to systems with big or infinite state space. Introducing more abstraction is the typical strategy to combat the state explosion problem. It is important for the abstract model to be sound, namely, that the properties proved on the abstraction are true about the original system. However, the abstract model is usually not complete – not all true properties of the original system are provable for the abstract model of a system [34].

Deductive verification is an approach which establishes the correspondence between a program and its specification by deductive reasoning [35]. The straightforward approach to deductive verification is to establish the meaning of a given program and then logically derive properties about its behaviour in universal logic (e.g., predicate logic). The downside of this approach is the need to manually model the program behaviour [4]. Clearly, it is error-prone and does not scale to larger software. More advanced approaches to deductive verification provide and justify a program logic for the semantics of a fixed programming language. This allows to automatically generate verification conditions which must be proved to establish that program satisfies its specification. The proof of verification conditions can be done manually or automatically (e.g., by using SMT solvers). The downside of the approach is that overall correctness depends on the soundness of the program logic used, the correctness of verification condition generator, and the correctness of used SMT solvers [4].

Dependently typed programming is an approach in which the specification, programming and proving can be done in one uniform environment. Type-checker establishes then the validity of a given proof that the program satisfies the specification. In this work, we use the dependently typed functional programming language Agda [38].

To guarantee that the compiled code behaves as it is prescribed by the semantics of the source code, every single phase of compilation process must be formally specified and verified. In Paper IV we used Agda to show how to implement a provably correct parser generator for regular languages (lexical analysis). The objective of this thesis is to develop a set of tools and a certified parser gener-

ator for context-free languages (syntax analysis) in the dependently typed setting of Agda.

1.3 Constructive logic

Constructive logic or intuitionism is a type of symbolic logic that differs from classical logic by replacing the traditional concept of truth with the concept of constructive provability [49]. In classical logic, propositions are always assigned a truth value—each proposition is either "true" or "false". It is not required to have an explicit proof for either case. In contrast, in constructive logic, propositions are not assigned any truth values. Instead a proposition is considered to be "true" only when we have an explicit proof. In constructive logic, proof comes before truth.

Unproved statements remain undecided until they are either proved or disproved. Statements are disproved by deducing a contradiction from them. Proof-theoretically, intuitionistic logic is a restriction of classical logic in which the law of excluded middle is not among the tautologies. The law of excluded middle can still be used for decidable propositions, but does not hold universally [49].

An important aspect of constructive logic is that because of its restrictions the proofs have the existence property. This means that, if there is a constructive proof that an object exists, then the proof may be used to produce an algorithm for generating that object [49].

The algorithmic content of proofs in the constructive setting makes this logic interesting to the programming language community. Surprisingly, it turns out that programs written in functional languages with rich type systems can be seen as constructive proofs of propositions corresponding to the types of programs [51].

1.4 Curry–Howard correspondence

The Curry–Howard correspondence is a relationship between computer programs and mathematical proofs. It was discovered by Haskell Curry and William Alvin Howard. However, the idea is related to the functional interpretation of intuitionistic logic given in various formulations by L. E. J. Brouwer, Arend Heyting and Andrey Kolmogorov (BHK interpretation), and Stephen Kleene [49].

The Curry–Howard correspondence is the observation that proof systems and models of computation are in fact structurally the same kind of objects. The main idea can be expressed as the following claim: a proof is a program, the formula it proves is a type for the program [51].

The claim states that the hypotheses of a logical theorem correspond to argument values passed to the function whereas the return type of a function corresponds to the statement of the theorem under these hypotheses. Therefore, proofs can be represented as programs which in turn can be run.

The correspondence is as follows:

Logic	Programming
universal quantification	dependent function type
existential quantification	dependent product (pair) type
implication	function type
conjunction	product (pair) type
disjunction	sum type
truth	unit type
falsity	empty type

For example, Haskell has the well-known operator `$`:

```
($) : (a -> b) -> a -> b
($) f a = f a
```

Interestingly, the type can be read as the well-known rule of implication elimination from predicate logic or as the type of a higher-order polymorphic function. An inhabitant (or proof) of this type (or rule) is given by function application.

The Curry–Howard correspondence led to a new kind of typed calculi designed to act both as a proof framework and as a functional programming language with expressive type system. Martin-Löf's type theory and Coquand's Calculus of Constructions are examples of such frameworks. These formal systems define a language in which proofs (functions) are first-class citizens. Therefore, it becomes possible to state properties of functions (proofs). These formalisms led to the development of software like Coq and Agda which allow one to use single formal language for writing types, specifications, functions, and proofs [39].

1.5 Basics of Agda

In this section, we show some simple function definitions and proofs that should give our reader a glimpse into programming and proving in the Agda system [1, 38].

Dependent types are types that can refer to values. Ordinary function types `X → Y` can refer only to other types. Dependent function types `(x : X) → Y` allow to give a name, `x`, to a typical element of the domain `X` of the function. The name `x` can be then used in the codomain `Y` to refer to the element of `X` that has been supplied as the argument of the function [12]. For example,

```
reflexivity : (X : Set) → (x : X) → x ≡ x
```

In the provided example, we first assume that `X` is a set, then we say that for any element `x` of a set `X` we can compute the value of the set `x ≡ x`, which is to be seen as the type of proofs that `x` equals to itself. Note, that the identity type depends on the set `X` (in an implicit or hidden way), but also on a particular element `x` of that set.

Agda supports a wide class of strictly positive inductive and inductive-recursive datatypes. Parametric inductive datatypes can be defined by providing a variable

declarations after the name of the datatype. For example, polymorphic lists have one parameter (`X : Set`):

```
data List (X : Set) : Set where
  []   : List X
  _::_ : X → List X → List X
```

This declares `List X` to be the set of lists of elements of type `X`. Since the definition is parameterized in a type `X`, we have effectively defined a family of types `List X`, one for each `X`.

To make the code more readable, in some cases we use the bracket notation to denote lists, i.e., `[ 1 , 2 ]`, instead of the basic notation `1 :: 2 :: []`.

Datatypes can also be indexed. In such cases, they are called inductive families. The difference between an index and a parameter is that the index need not be constant throughout the definition of the type [12]. For example, consider the type `x ∈ xs` of proofs that `x` is the element of list `xs`. Consider the definition:

```
data _∈_ {X : Set}(x : X) : List X → Set where
  here  : {xs : List X} → x ∈ x :: xs
  there : {y : X}{xs : List X} → x ∈ xs → x ∈ y :: xs
```

In Agda, an argument enclosed in curly braces is implicit. The Agda type-checker will try to figure it. If an argument cannot be inferred, it must be provided explicitly. So, the definition is implicitly parameterized by the type of elements (`X : Set`), explicitly parameterized by the element of that type (`x : X`) and is indexed by list of elements of `X`. Therefore, we can naturally express that either the element is in the head position of a list `x ∈ x :: xs` or, if we know that `x ∈ xs`, then it is also true that `x ∈ y :: xs` for any `y`.

The main idea behind the Curry–Howard isomorphism is that each proposition can be viewed as the set of its proofs. To emphasize that proofs here are first-class mathematical objects, one often talks about "proof objects". In constructive mathematics, they are sometimes also referred to as constructions. A proposition is considered to be true, if its set of proofs is inhabited; it is false, if its set of proofs is empty. The canonical empty type is defined as follows:

```
data ⊥ : Set where
```

It is clear that it is impossible to have an element of $\bot$, as there are no constructors.

As usual in constructive logic, to prove the negation of a proposition is the same as proving that the proposition in question leads to absurdity:

```
¬_ : Set → Set
¬ X = X → ⊥
```

Moreover, we have the following elimination principle:

```
ex-falso-quodlibet : {X : Set} → ⊥ → X
```

A proof by (well-founded) induction is a recursive function with the requirement that it needs to be terminating on all inputs. Termination is checked by Agda using the so-called structural ordering on datatypes.

In our final example, we would like to demonstrate a proof that, for any type `X` for which equality is decidable, we can also decide the list membership relation. The equality type itself (already used above) is an inductive type defined as follows:

```
data _≡_ {X : Set} (x : X) : X → Set where
  refl : x ≡ x
```

A decider for equality is a function of two arguments of some type `X` which returns either a proof that the arguments are equal or a proof that equality of the arguments provided implies absurdity.

```
DecEq X = (x₁ x₂ : X) → x₁ ≡ x₂ ⊎ ¬ x₁ ≡ x₂
```

Next, we will develop our proof in a step-by-step fashion. The theorem we want to prove is expressed by the type:

```
∈-dec : {X : Set} → DecEq X → (x : X) → (xs : List X)
                                      → x ∈ xs ⊎ ¬ x ∈ xs
```

We start by pattern matching on the given list `xs`. Clearly, it can be empty or not. If the list is empty, then we conclude that `x` is not its member. Technically, this is done by choosing the second alternative from the sum type ($\mathtt{inj}_2$) and then proving that the type `x ∈ []` is empty. When there are no possible constructor patterns for a given argument (as in case with `x ∈ []`), one can pattern match on it with the absurd pattern `()`.

```
∈-dec deq x [] = inj₂ (λ ())
∈-dec deq x (x' :: xs') = ?
```

The next step is to analyze the case when the list `xs` has head `x'` and tail `xs'`. Then we use the given decidable equality to check whether `x` is equal to `x'`.

```
∈-dec deq x (x' :: xs') with deq x x'
∈-dec deq x (x' :: xs') | inj₁ p = ?
∈-dec deq x (x' :: xs') | inj₂ p = ?
```

We again branch into two cases. In the first case, `x` is equal to `x'` and `p` is a proof of `x ≡ x'`. Therefore, we can pattern match on `p` and, since the only possible canonical proof of `x ≡ x'` is `refl`, we can substitute `x` for `x'` everywhere. Then the goal of this branch becomes `x ∈ x :: xs` and this is exactly what the constructor `here` gives.

```
∈-dec deq x (x' :: xs') with deq x x'
∈-dec deq .x (x :: xs') | inj₁ refl = inj₁ here
```

The notation `.x` indicates that this argument is forced to be `x`, given that the equality proof is assumed to be `refl`.

In the second case, `x` is not equal to `x'`. Then we must recursively check whether `x` is a member of the tail `xs'`. We do that by recursively calling `∈-dec deq x xs'`. Since the argument list becomes shorter, the termination checker is satisfied. The recursive call branches into cases where `x` is a member of the tail or not. If `x` is a member of the tail, then we must prove `x ∈ x' :: xs`, given that `x ∈ xs`. This is done using the constructor `there`:

```
∈-dec deq x (x' :: xs') | inj₂ p with ∈-dec deq x xs
∈-dec deq x (x' :: xs') | inj₂ p | inj₁ x₁ = inj₁ (there x₁)
```

Otherwise, we know that `p : ¬ x ≡ x'` and `q : ¬ x ∈ xs'`. This is enough to prove that the type `x ∈ (x' :: xs')` is empty.

```
∈-dec deq x (x' :: xs') | inj₂ p | inj₂ q
  = inj₂ (λ { here pr → p pr ; there pr → q pr })
```

The last proof term demonstrates the use of pattern matching under lambdas.

1.5.1 Intrinsic vs. extrinsic styles

When implementing a program in a dependently typed language, there is a design choice between "extrinsic" and "intrinsic" verification. In case of the extrinsic style, the developer implements simply typed functions and datastructures and then proves properties about their behaviour. For example, we can define a function `length` on lists:

```
length : {X : Set} → List X → ℕ
length [] = 0
length (_ :: xs) = 1 + length xs
```

Next, we want to show the relation between concatenation and the length of lists:

```
length-lemma : {X : Set} → (xs ys : List X)
     → length (xs ++ ys) ≡ length xs + length ys
```

In case of intrinsic development, the developer encodes the properties into the types of datastructures and functions. Instead of working with simply typed lists, we can use vectors, which are lists indexed by their lengths:

```
data Vec (X : Set) : Nat → Set  where
   []  : Vec X zero
  _::_ : {n : Nat} (x : X) → (xs : Vec X n) → Vec X (suc n)
```

Now, the type signature of the concatenation function is forced to make explicit the size of a resulting vector:

```
_++_ :: {n m : Nat}{X : Set} → Vec X n → Vec X m
     → Vec X (n + m)
```

In our work, we use both styles of development. The extrinsic approach makes it easier to separate executable functions from proofs of correctness, which makes it simpler for the compiler to optimize the resulting code. On the other hand, the intrinsic style allows one to design datastructures and functions which are correct-by-construction and therefore significantly ease and shorten the proofs. This becomes even more important, since Agda provides only limited ways to automatization.

1.5.2 Compilation

Agda has a compiler that can extract Haskell or JavaScript. The Agda's extraction mechanism is still being developed and there are significant improvements in recent versions (especially comparing older version to Agda 2.5.1). Indeed, we observed a significant improvement in performance when compiling our development with Agda 2.5.1 versus Agda 2.4.*. This seems to be the result of the numerous adjustments in compilation of Agda (the exact details can be found in the release notes to version 2.5.1).

We believe that larger implementations (like ours) can help Agda developers to make the extraction mechanism better.

Based on our experiments, we do not think that at the current stage the Agda extraction mechanism generates code with competitive performance.

1.5.3 Automatization

Agda provides a term search tool Agsy. The tool is independent of Agda and any solution coming from it is checked by Agda. However, one should not expect it to handle large problems of any particular kind. It is meant more as a convenience tool during the development process.

Due to the absence of automatization in Agda comparable to tactics in Coq, the proofs can end up being verbose. On the other hand, it forces the developer to design the datastructures and datatypes more carefully, so that proofs remain manageable in reuse and generalization.

2 Finite sets in dependently typed setting

Intuitionistic frameworks give rise to a rich variety of competing notions of finiteness that collapse classically. Therefore it is important to know the exact relationships between the some variations. From the programming perspective, the most important definition of finiteness is listability: a set is considered to be finite, if its elements can be completely enumerated in a list. In this section, we describe the main results from Paper I, where we formalize different variations of listability in the Agda programming language and develop a toolbox for boilerplate-free programming with finite sets that are given by listing a subset of some base set with decidable equality. The resulting library can, in particular, be used to represent, manipulate and derive properties of context-free grammars.

In Section 2.1, we give the basic definition of a listable set and some variations of this definition. We proceed by showing that listability implies decidable equality and use this fact to show that the notions of finiteness introduced are equivalent. In Section 2.2, we define the notion of a listable subset given by a predicate on some base set and prove that it is not possible to generally derive decidability of equality for listable subsets. Next, in Section 2.3, we describe an approach to concise definition of new listable sets by using squashing of decidable properties. In Section 2.4, we implement some combinators for defining functions on finite sets. Finally, in Section 2.5, we develop a prover for quantified formulas over decidable properties on finite sets.

The Agda development corresponding to the content of this section is located in the module `Fin` of the accompanying code.

2.1 Listable sets

The simplest and strongest notion of finiteness in the constructive setting is listability [31]. A set is listable, if there exists a list containing all of its elements:

```
All : (X : Set) → List X → Set
All X xs = (x : X) → x ∈ xs

Listable : (X : Set) → Set
Listable X = ∃[ xs : List X ] All X xs
```

The type `All X xs` tells us that the list `xs` contains all elements of type `X`. The type `Listable X` says that there exists a list `xs` with the property `All X xs`.

Alternatively, we can ask for a surjection from an initial segment of natural numbers. The type `Fin n` is an inductive type containing all natural numbers strictly smaller than `n`.

```
FinSurj : (X : Set) → Set
FinSurj X = ∃[ n : N ]
            ∃[ fromFin : Fin n → X ]
```

```
              ∃[ toFin : X → Fin n ]
              ((x : X) → fromFin (toFin x) ≡ x)
```

So, the proof of `FinSurj X` contains the size `n` of an initial segment, a function `fromFin` from `Fin n` to `X`, a function `toFin` from `X` to `Fin n` and a proof that `fromFin` is a surjection, i.e., that `toFin` is a pre-inverse of `fromFin`. We also prove that the two notions are equivalent:

```
surj2lstbl : {X : Set} → FinSurj X → Listable X

lstbl2surj : {X : Set} → Listable X → FinSurj X
```

To implement `surj2lstbl`, we get the size `n` of the initial segment from the argument `FinSurj X` and then generate a list `xs` by applying `fromFin` to the first `n` natural numbers. The proof of `All X xs` is implied by the fact that `fromFin` is a surjection. Implementing `lstbl2surj` requires us only to associate every position in the list `xs` with the element at this position. The rest follows.

We can try to define a stronger notion by additionally asking for a proof that all elements of a set appear in the list only once.

```
ListableNoDup : (X : Set) → Set
ListableNoDup X = ∃[ xs : List X ]
                  All X xs ×
                  NoDup xs
```

Since all positions in the list `xs` are in an one-to-one correspondence with elements of `X`, it is the same as asking for a bijection from an initial segment of the set of natural numbers:

```
FinBij : (X : Set) → Set
FinBij X = ∃[ n : N ]
           ∃[ fromFin : Fin n → X ]
           ∃[ toFin : X → Fin n ]
           ((x : X) → fromFin (toFin x) ≡ x) ×
           ((i : Fin n) → toFin (fromFin i) ≡ i)

bij2lstblnd : {X : Set} → FinBij X → ListableNoDup X

lstblnd2bij : {X : Set} → ListableNoDup X → FinBij X
```

In Section 3 of Paper I, we show that all four introduced notions of finiteness are of the same strength. The reason for this is that we can derive decidability of equality for elements of type `X`, if it is listable:

```
DecEq X = (x₁ x₂ : X) → x₁ ≡ x₂ ⊎ ¬ x₁ ≡ x₂

lstbl2eq : {X : Set} → Listable X → DecEq X
```

The idea for deriving decidability of equality for elements of type `X` from the proof of `Listable X` is as follows. Given elements x_1 and x_2, we can learn their positions p_1 and p_2 in the list `xs` by using the proof of `All X xs`. If the positions are equal, then clearly the elements x_1 and x_2 are also equal. If the positions are different, then we can show that the elements must also be different. Assume the opposite. Then $x_1 \equiv x_2$ implies $p_1 \equiv p_2$, since the proof of `All X xs` is a function and must return equal results for equal arguments. We have derived a contradiction from the assumption that the positions are different.

With decidable equality at our disposal, it is routine to implement a function which removes duplicates from the list and generates a proof that the resulting list is duplicate free (`NoDup xs`). Finally, we can show that `Listable X` is as strong as `ListableNoDup X`:

```
lstbl2lstblnd : {X : Set} → Listable X → ListableNoDup X

lstblnd2lstbl : {X : Set} → ListableNoDup X → Listable X
```

2.2 Listable subsets

A special case of sets are those defined as a subset of a larger set. Here, by a subset we really mean a description of a subset by means of a base set and a predicate on it. We do not consider a subset type former. A subset of a base set `U` carved out by a predicate `P : U → Set` is finite, if there is a list containing all elements of `U` that satisfy `P` and only these:

```
ListableSub : (U : Set) → (U → Set) → Set
ListableSub U P = ∃[ xs : List U ]
                  ((x : U) → P x → x ∈ xs) ×
                  ((x : U) → x ∈ xs → P x)
```

Listable sets are a special case of listable subsets:

```
lstbl2lsub : {U : Set}
  → Listable U → ListableSub U (λ _ → ⊤)

lsub2lstbl : {U : Set}
  → ListableSub U (λ _ → ⊤) → Listable U
```

(Here $\top$ is the unit type: a singleton type with a unique element `tt`.)

Next, we establish that the notion of listable subsets is more general than the notion of listable sets. We observe that it is impossible to derive a decidable equality from a proof of `ListableSub U P` for elements of set `U` which satisfy `P`. In Section 3 of Paper I, we show that having decidable equality for the general case would imply functional extensionality, which is unavailable in the type theory Agda is based on.

However, there are two special cases when decidability of equality becomes provable. The first is when there is at most one proof of `P x` for every `x`. The strategy of deriving the decidable equality for elements of such subset is same as the one described for listable sets. The crucial point here is that, since, for each element `x`, there is only one proof of `P x`, then the "completeness" proof

```
(x : U) → P x → x ∈ xs
```

cannot provide different proofs of `x ∈ xs` for different proofs of `P x`.

The second special case when equality becomes decidable for elements of a listable subset happens when the predicate `P` is decidable. This case is actually reducible to the first case by using effective squash types:

```
‖_‖ : {Q : Set} → Dec Q → Set
‖ yes _ ‖ = ⊤
‖ no  _ ‖ = ⊥
```

If a set `Q` is decidable, we can effectively define a squashed version of `Q` (i.e., the quotient `Q` by the total equivalence relation). Next, we observe that, if a predicate `P` is decidable, namely, there exists a decider `dec : ∀ x → Dec (P x)`, then we can define a predicate `P' : (x : X) → ‖ dec x ‖` so that, for any `x` from `X`, `P x` is equivalent to `P' x` and moreover there is a unique proof of `P' x` for any `x`.

It is also important to note that one can always get a proof of `P`, if the squashed version is inhabited:

```
fromSq : {P : Set} → (d : Dec P) → {‖ d ‖} → P
```

We have made the third argument (of type `‖ d ‖`) implicit, since, if `d` proves `Dec P`, then the only possible value is the unique element `tt : ⊤` and the type-checker can derive it automatically.

2.3 Pragmatic finite subsets

Direct construction of (new) listable sets requires tedious manual work in Agda. In this section, we describe an approach to specifying new listable sets as subsets of some base set with decidable equality. This approach allows one to specify a listable set by listing the selected elements of the base set once. Moreover, the decider of equality and the proof of listability of the newly defined set are generated automatically.

We start by defining a new type `FinSubDesc` which is parameterized by some base set `U`, a decidable equality on its elements, and a Boolean flag. Values of this type are descriptions of new finite sets.

```
FinSubDesc : (U : Set) (eq : DecEq U) → Bool → Set
```

To specify a description of a finite subset there are two constructors: `fsd-plain` and `fsd-nodup`. The first of those takes a list of elements of type `U` and returns a description of a finite subset of type `FinSubDesc U eq true`. The second constructor takes a list of elements of type `U` and a proof that the list contains no duplicates and returns a value of type `FinSubDesc U eq false`.

The Boolean flag indicates whether the underlying list of the description is allowed to have duplicates. This information can be exploited for better performance when implementing combinators on descriptions of finite sets.

The type of elements of pragmatic finite subsets is parameterized by such description of a finite set:

```
Elem : {U : Set}{eq : DecEq U}{b : Bool}
  → FinSubDesc U eq b → Set
Elem {U} {eq} D = ∃[ x : U ] ∥ x ∈? toList D ∥
  where
    _∈?_ = ∈-dec eq
```

So, an element of type `Elem D` for some finite subset description `D` is a dependent pair of an element `x` of `U` together with a squashed proof that `x` belongs to the list of elements defining the subset. Using the squashed membership type allows us to ignore the exact position(s) of the element in the list.

An element of type `Elem D` is thus a pair `(x , p)`. However, the squashed membership proof `p` is of little interest for most of the time. Therefore, we introduce the alternative notation for elements of type `Elem D`:

```
⟨_⟩ :  {U : Set}{eq : DecEq U}{b : Bool}
  → {D : FinSubDesc U eq b} → (x : U)
  → {p : ∥ x ∈? toList D ∥} → Elem D
⟨ x ⟩ {p = p} = (x , p)
```

When we write `⟨ x ⟩`, we assume that there is enough information in the context, so that the typechecker can infer all the implicit arguments.

The most important property is that, for all `D : FinSubDesc U eq b`, the corresponding subset of `U`, namely `Elem D`, is listable.

First, we generate a list of elements of `Elem D`:

```
listElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (D : FinSubDesc U eq b)
  → List (Elem D)
```

Second, we show that `listElem D` is complete, it contains all elements of `Elem D`:

```
allElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (D : FinSubDesc U eq b)
  → (xp : Elem D) → xp ∈ listElem D
```

These two properties together give us that `Elem D` is listable and has decidable equality:

```
lstblElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (D : FinSubDesc U eq b)
  → Listable (Elem D)

deqElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (D : FinSubDesc U eq b)
  → DecEq (Elem D)
```

2.4 Functions on finite sets

Functions on finite sets can be represented as lists of tuples. In Agda, we can specify a type `Tbl` parameterized by a domain type and a codomain type:

```
Tbl : Set → Set → Set
Tbl X Y = ∃[ xys : List (X × Y) ]
          All X (map proj₁ xys) ×
          NoDup (map proj₁ xys)
```

A value of type `Tbl X Y` is a list of pairs of type `X × Y` with some additional information, namely, a proof that the list of pairs is complete, i.e., each element of `X` is associated with a value of `Y`. Additionally, the list must be duplicate-free regarding the first components. The first property ensures totality while the second guarantees unambiguous interpretation. Also, recall that `All X xs` implies `Listable X`.

We prove that any table can be converted into a function:

```
fromTbl : {X Y : Set} → Tbl X Y → X → Y
```

Conversely, if `X` is listable then any function `X → Y` can be represented as a table:

```
toTbl : {X Y : Set} → Listable X → (X → Y) → Tbl X Y
```

The correctness condition says that the function `f` and the result of converting `f` to the table and back are extensionally equal:

```
fromto : {X Y : Set} → (p : Listable X) → (f : X → Y)
  → (x : X) → fromTbl (toTbl p f) x ≡ f x
```

For larger finite sets, the approach of defining functions in terms of tables is tedious and error-prone. Therefore, we propose a notion of predicate matching. We start by implementing a function `unreached` that takes a list of decidable predicates and a list of elements and returns the list of those predicates that are not satisfied by any element:

```
unreached : {X : Set} → List (X → Bool)
  → List X → List (X → Bool)
```

Soundness If, for some list xs, the list of predicates ps contains no unreachable ones, then for any split of ps into three parts, ps $\equiv$ ps$_1$ ++ p :: ps$_2$, there exists at least one element x that satisfies p but does not satisfy any of the predicates in ps$_1$.

Completeness On the other hand, for any split ps $\equiv$ ps$_1$ ++ p :: ps$_2$ of the list of predicates ps, if all elements of a list xs which do not satisfy any predicates from ps$_1$ also do not satisfy p, then p is unreachable.

Now, let us address the issue of unmatched elements. We implement a function unmatched that returns the list of all those elements in a given list xs that do not satisfy any predicate in the given list ps:

```
unmatched : {X : Set} → List (X → Bool) → List X
  → List X
```

Soundness If there are no unmatched elements in the list xs, then, for any element x in xs, the list of predicates ps can be split into three parts, ps $\equiv$ ps$_1$ ++ p :: ps$_2$, so that no predicate from ps$_1$ is satisfied by x and p is satisfied by x.

Completeness If each element in the list xs satisfies at least one predicate in ps, then there are no unmatched elements.

We can now define a combinator that takes a list of predicates associated with the functions and the proofs that all elements are matched and all predicates reached, and returns a function built from the pieces:

```
predicateMatching : {X Y : Set}
  → (ps : List ((X → Bool) × (X → Y)))
  → (p : Listable X)
  → unmatched (map proj₁ ps) (proj₁ p) ≡ []
  → unreached (map proj₁ ps) (proj₁ p) ≡ []
  → X → Y
```

2.5 Prover

Conceptually, it is easy to see that existential and universal statements over finite sets are decidable for decidable properties. For example, to prove that a decidable predicate Q holds for all elements of a finite set U, we need to check it on each element. Therefore, the notion of listable finiteness suits well.

We implement a combinator subAll? that takes some decidable property Q on elements of U and returns a decision of whether Q holds for all elements of the listable subset defined by predicate P.

```
subAll? : {U : Set}{P : U → Set}
```

```
  → ListableSub U P
  → {Q : U → Set}
  → ((x : U) → {P x} → Dec (Q x))
  → Dec ((x : U) → {P x} → Q x)
```

The same can be done for the existential quantifier:

```
subAny? : {U : Set}{P : U → Set}
  → ListableSub U P
  → {Q : U → Set}
  → ((x : U) → {P x} → Dec (Q x))
  → Dec (∃[ x : U ] P x × Q x)
```

The combinators `subAll?` and `subAny?` are sufficient to decide properties which are in prenex form with the quantifiers ranging over the whole finite subset given by `P`.

Finally, we provide some syntactic sugar for our combinators:

```
syntax subAll? f (λ x → z) = Π x ∈ f , z
syntax subAny? f (λ x → z) = ∃ x ∈ f , z
```

(Agda will automatically rewrite expressions matching the right-hand side into the corresponding terms on the left.)

2.6 Example

We illustrate the advantages of using the definitions described by comparing them to the straightforward approach of defining and working with finite sets from datatypes with only nullary constructors (i.e., enumeration types). In what follows, we define an example context-free grammar by following the definition given in the introductory section.

We start by defining sets for nonterminals and terminals. In most of the cases these sets could be chosen as some built-in types like `Char` or `String`. In this example, we define dedicated finite types for nonterminals and terminals to illustrate the verbose repeating patterns in the straightforward definitions:

```
data N : Set where
  A : N
  B : N
  C : N

data T : Set where
  1 : T
  0 : T
```

The grammar rules can then be defined as an inductive family indexed by the left and right hand sides:

```
data _⟶_ : N → List (N ⊎ T) → Set where
  rule1 : A ⟶ [ inj₁ C , inj₁ B ]
  rule2 : B ⟶ [ inj₁ C , inj₁ B ]
  rule3 : C ⟶ [ inj₂ 1 ]
```

The set of all rules is then a dependent pair type. We also provide projections for the left and right hand sides:

```
Rule : Set
Rule = ∃[ L : N ] ∃[ R : List (N ⊎ T) ] L ⟶ R

projL : Rule → N
projL = proj₁

projR : Rule → List (N ⊎ T)
projR r = proj₁ (proj₂ r)
```

To show that the sets defined are finite (so that one can, for example, iterate through all elements), we can provide lists:

```
listN : List N
listN = [ A , B , C ]

listT : List T
listT = [ 0 , 1 ]

listRule : List Rule
listRule = [ (_ , _ , rule1) , (_ , _ , rule2)
                             , (_ , _ , rule3) ]
```

(The underscore _ is the "don't care, go figure it out" pattern which tells the type-checker that there should be enough information for it in the context to uniquely determine the exact terms at the underscored places.)

Then we can prove that the given lists are complete:

```
allN : (n : N) → n ∈ listN
allN A = here
allN B = there here
allN C = there (there here)

allT : (t : T) → t ∈ listT
allT 0 = here
allT 1 = there here

allRules : (r : Rule) → r ∈ listRule
```

```
allRules (_ , _ , rule1) = here
allRules (_ , _ , rule2) = there here
allRules (_ , _ , rule3) = there (there here)
```

Next, we also need a equality decider for the elements of the sets defined:

```
_=N?_ : DecEq N
A =N? A = inj₁ refl
A =N? B = inj₂ (λ ())
A =N? C = inj₂ (λ ())
B =N? A = inj₂ (λ ())
B =N? B = inj₁ refl
B =N? C = inj₂ (λ ())
C =N? A = inj₂ (λ ())
C =N? B = inj₂ (λ ())
C =N? C = inj₁ refl

_=T?_ : DecEq T
0 =T? 0 = inj₁ refl
0 =T? 1 = inj₂ (λ ())
1 =T? 0 = inj₂ (λ ())
1 =T? 1 = inj₁ refl

_=R?_ : DecEq Rule
(_ , _ , rule1) =R? (_ , _ , rule1) = inj₁ refl
(_ , _ , rule1) =R? (_ , _ , rule2) = inj₂ (λ ())
(_ , _ , rule1) =R? (_ , _ , rule3) = inj₂ (λ ())
(_ , _ , rule2) =R? (_ , _ , rule1) = inj₂ (λ ())
(_ , _ , rule2) =R? (_ , _ , rule2) = inj₁ refl
(_ , _ , rule2) =R? (_ , _ , rule3) = inj₂ (λ ())
(_ , _ , rule3) =R? (_ , _ , rule1) = inj₂ (λ ())
(_ , _ , rule3) =R? (_ , _ , rule2) = inj₂ (λ ())
(_ , _ , rule3) =R? (_ , _ , rule3) = inj₁ refl
```

We also prove that each right hand side is either a single terminal or a pair of nonterminals. Moreover, we show that the terminal `A` never appears on the right hand side of any rule:

```
RHS-prop : (r : Rule) → ∃[ t : T ] (projR r ≡ [ inj₂ t ] ) ⊎
         ∃[ B : N ] ∃[ C : N ] (projR r ≡ [ inj₁ B , inj₁ C ])
RHS-prop (_ , _ , rule1) = inj₂ (C , B , refl)
RHS-prop (_ , _ , rule2) = inj₂ (C , B , refl)
RHS-prop (_ , _ , rule3) = inj₁ (1 , refl)
```

```
S-prop : (r : Rule) → ¬ inj₁ A ∈ projR r
S-prop (_ , _ , rule1) (there (there ()))
S-prop (_ , _ , rule2) (there (there ()))
S-prop (_ , _ , rule3) (there ())
```

The proofs are done by complete analyses of cases.

As we will see later, the combination of the definitions provided specifies a context-free grammar in Chomsky normal form with start nonterminal `A`.

Next, we will try to define the same grammar by using finite sets defined as subsets of the type `Char`.

First, we define descriptions of the sets of terminals and nonterminals by listing their elements. These descriptions induce sets for which listability and decidable equality are derived automatically:

```
N-Desc : FinSubDesc Char _=C?_ false
N-Desc = fsd-nodup [ 'A' , 'B' , 'C' ]

N = Elem N-Desc

listableN : Listable N
listableN = lstblElem N-Desc

_=N?_ : DecEq N
_=N?_ = deqElem N-Desc

T-Desc : FinSubDesc Char _=C?_ false
T-Desc = fsd-nodup [ '0' , '1' ]

T = Elem T-Desc

listableT : Listable T
listableT = lstblElem T-Desc

_=T?_ : DecEq T
_=T?_ = deqElem T-Desc
```

The operator `_=C?_` is a decider of equality for the built-in type `Char`. Next, the set of rules is given as a subset of the set `N × List (N ⊎ T)`. And again, the proofs of listability and decidable equality are derived:

```
_=RB?_ : DecEq (N × List (N ⊎ T))

_⟶_ : {A B : Set} → A → B → A × B
_⟶_ = _,_
```

```
Rule-Desc : FinSubDesc N × List (N ⊎ T) _=RB?_ false
Rule-Desc = fsd-nodup [ ⟨'A'⟩ ⟶ [ inj₁ ⟨'C'⟩ , inj₁ ⟨'B'⟩ ]
                      , ⟨'B'⟩ ⟶ [ inj₁ ⟨'C'⟩ , inj₁ ⟨'B'⟩ ]
                      , ⟨'C'⟩ ⟶ [ inj₂ ⟨'1'⟩ ] ]

Rule = Elem Rule-Desc

projL (r , _) = proj₁ r
projR (r , _) = proj₂ r

listableRule : Listable Rule
listableRule = lstblElem Rule-Desc

_=R?_ : DecEq Rule
_=R?_ = deqElem Rule-Desc
```

(_⟶_ is a synonym for the product type former.) The `Rule-Desc` uses the equality decider `_=RB?_` of elements of `N × List (N ⊎ T)` which is easily derived from equality deciders of `N` and `T`.

By using the prover combinators, the proofs of `RHS-prop` and `S-prop` amount essentially to just restating the properties:

```
_=RHS?_ : DecEq List (N ⊎ T)

RHS-prop : (r : Rule) → ∃[ t : T ] (projR r ≡ [ inj₂ t ] ) ⊎
          ∃[ B : N ] ∃[ C : N ] (projR r ≡ [ inj₁ B , inj₁ C ])
RHS-prop = fromSq (
   (Π r ∈ listableRule ,
    ∃ t ∈ listableT , projR r =RHS? [ inj₂ t ])
   ⊎-dec
   (Π r ∈ listableRule ,
    ∃ B ∈ listableN ,
    ∃ C ∈ listableN , projR r =RHS? [ inj₁ B , inj₁ C ]))


S-prop : (r : Rule) → ¬ inj₁ ⟨'A'⟩ ∈ projR r
S-prop = fromSq (Π r ∈ listableRule , (inj₁ ⟨'A'⟩) ∈? projR r)
```

The combinator `⊎-dec` establishes that, if we can decide `P a` and `Q a` for every `a`, then `P a ⊎ Q a` is also decidable. The operator `_=RHS?_` is a decider of equality for `List (N ⊎ T)`, which can be trivially derived from the equality deciders of `N` and `T`.

2.7 Conclusions

In this section, we studied listability of sets in the constructive setting. We observed that listability (i.e., that a set can be completely enumerated in a list) implies decidable equality and that certain simple variations of this definition are all equivalent. Next, we explained how squashing of decidable predicates allowed us to implement a library of combinators for programming with listable sets. Using the library, the programmer has never to supply list membership proofs and the combinators are able to traverse the enumerating list of the set to check decidable properties of its elements. We also described an approach to concise definition of new listable sets.

We will use this library in the next section to deal with finite sets of terminals, nonterminals, and production rules.

3 CYK parsing of context-free languages

In this section (following Paper II), we implement in Agda the parsing technique which is attributed to J. Cocke, D. H. Younger, and T. Kasami, who independently discovered variations of the method known as the Cocke–Younger–Kasami algorithm (CYK).

The CYK algorithm [52] belongs to the family of bottom-up methods which grow a parse tree from the input string up towards the start nonterminal. This algorithm works only on context-free grammars in Chomsky normal form.

From among many other parsing algorithms, we decided to focus on the CYK method because of its simple and elegant structure. The descriptions found in literature usually divide CYK parsing into two phases. The first phase of the algorithm constructs a table establishing which nonterminals derive which substrings of the given string. This is the recognition phase, which also determines whether the input string can be derived from the start nonterminal. The second phase uses the recognition table and the grammar to construct derivations of the string's substrings. We digress slightly from the classical approach and combine the two phases into one by using tables over sets of parse trees.

Sections 3.1 and 3.2 present the definitions of a context-free grammar in Chomsky normal form, the parsing relation, and the naive version of the CYK algorithm. In Section 3.3, we show that the algorithm is correct. Section 3.4 is devoted to discharging the obligations of the termination checker. Finally, Section 3.5 reports how memoization can be introduced systematically into the naive implementation, yielding the intended performance while retaining the correctness guarantee.

The Agda development corresponding to the content of this section is located in the module `CYK` of the accompanying code.

3.1 Context-free grammars and parsing relation

In this section, we slightly digress from our previous approach to defining context-free grammars. Initially, in Paper II, we modeled finite sets of terminals, nonterminals and productions with lists. Now we want to work with listable sets as this approach allows to use the full power of the previously developed combinators for finite sets.

We start by giving a fully formal definition of a grammar. We assume some sets for `N` and `T` for nonterminals and terminals respectively. Both types must come with decidable equality:

```
_=N?_ : DecEq N

_=T?_ : DecEq T
```

The rules of a grammar are specified by a finite subset of `N × List (N ⊎ T)`. So, before giving a formal definition of a grammar we provide some handy synonyms:

```
nt = N → N ⊎ T
nt = inj₁

tm : T → N ⊎ T
tm = inj₂

String = List T
RHS = List (N ⊎ T)

RB = N × RHS
```

Moreover, decidable equalities `_=N?_` and `_=T?_` induce decidable equality on `RB` (denoted by `_=RB?_`).

A context-free grammar in Chomsky normal form is a record of type `GrammarCNF` specifying the rules as the description of a subset of the set `RB`, a Boolean flag `nullable` indicating whether the language of the grammar contains the empty string, and a nonterminal `S` as the start nonterminal. Moreover, the record also contains proofs attesting normality of the grammar, i.e., that the nonterminal `S` never appears on the right hand side of a rule and that the right hand side of any rule is either a single terminal or pair of nonterminals.

```
record GrammarCNF : Set₁ where
  field
    Rule-Desc : FinSubDesc RB _=RB?_ true
    S : N
    nullable : Bool

    RHS-prop : (r : Rule)
       → ∃[ t : T ] (projR r ≡ [ tm t ]) ⊎
          ∃[ B : N ] ∃[ C : N ] (projR r ≡ [ nt B , nt C ])
    S-prop : (r : Rule) → nt S ∉ projR r

  Rule : Set
  Rule = Elem Rule-Desc

  Rs : Listable Rule
  Rs = lstblElem Rule-Desc

  _=R?_ : DecEq Rule
  _=R?_ = deqElem RuleDesc

  projL : Rule → N
  projL (r , _) = proj₁ r
```

```
proj_R : Rule → RHS
proj_R (r , _) = proj_2 r
```

Moreover, the definition of `GrammarCNF` defines a derived field `Rule` as the set of all elements of subset described by `Rule-Desc`, a derived field `Rs` as the proof of listability of `Rule`, and also projections for the left and right hand sides of the rules.

To avoid notational clutter, we denote a rule `r` as `A ⟶ rhs` if `proj_L r ≡ A` and `proj_R r ≡ rhs`. The rules for which `proj_R r ≡ [ nt B, nt C ]` will be denoted as `A ⟶ B , C`. The rules for which `proj_R r ≡ [ tm t ]` will be denoted as `A ⟶ t`. According to `RHS-prop`, these are the only possible shapes of rules allowed in `GrammarCNF`.

In this section, we assume one fixed grammar `G` in the context, so we use the fields of the grammar record directly, e.g., `S` and `nullable` instead of `S G` and `nullable G` for some given `G`.

Next, for any given string `s` we define the parsing relation of its substrings as an inductive predicate on two naturals.

```
data _[_,_)►_ (s : String) : ℕ → ℕ → N → Set where
  ...
```

The proposition `s [ i, j )► A` states that the substring of `s` from the `i`-th position (inclusive) to the `j`-th (exclusive) is derivable from nonterminal `A`. Proofs of this proposition are parse trees. The parsing relation has three constructors:

1. If the `nullable` flag is set, then the constructor `empt` constructs a parse tree for the empty string, namely, for any `i` we have `s [ i, i )► S`.

2. If the grammar contains a rule `A ⟶ a` for some nonterminal `A` and a terminal `a`, then given that the `i`-th position of a string `s` is occupied by `a`, the constructor `sngl` builds a parse tree for the substring between `i` and `suc i`, i.e. we have `s [ i, suc i )► A`.

3. If $\texttt{t}_1$ is a derivation of the substring between positions `i` and `j` starting from nonterminal `B`, $\texttt{t}_2$ is a parse tree for the substring between `j` and `k` from nonterminal `C` and the grammar contains a rule `A ⟶ B , C`, then the constructor `cons` combines those trees into a derivation of the substring between `i` and `k` starting from `A`.

According to item 1 above, the start nonterminal may be used to construct a parse tree for the empty string. Therefore it is crucial that each grammar comes with a proof `S-prop` that the start nonterminal never appears on the right hand sides of any rule. As a consequence of this restriction, every string in the language of the grammar has only finitely many parse trees.

3.2 CYK parsing algorithm

The CYK algorithm first concentrates on substrings of the input string, the shortest substrings first, and then works its way up. The derivations of substrings of length 1 can be read directly from the grammar, i.e., they are given by rules of shape `A ⟶ a`.

Next, if we are about to parse a string `s` of a longer length `l`, then we know that at the root of the parse tree a rule of the shape `A ⟶ B , C` must be used. Moreover, `B` has to derive the first part (which is non-empty) and `C` the rest (also non-empty). Therefore `B` must derive `s [ 0, k )▶ B`, likewise `C` must derive `s [ k, l )▶ C`. Determining whether such a `k` exists is easy: just try all possibilities from the range from `1` to `l - 1`.

To start implementing this algorithm in Agda, we introduce a new type `Mtrx` which is parameterized by a string `s`:

```
Mtrx : String → Set
Mtrx s = List (∃[i : ℕ] ∃[j : ℕ] ∃[A : N] s [ i, j )▶ A)
```

The definition says that `Mtrx s` is a list of all possible parse trees of substrings of `s`. Since each tree is parameterized by two natural numbers, `Mtrx s` is a linear representation of a two-dimensional matrix, hence the name. Next, we define the product of two matrices:

```
_*_  : {s : String} → Mtrx s → Mtrx s → Mtrx s
m₁ * m₂ = { (i, k, A, cons t₁ t₂) | (i, j, B, t₁) ← m₁,
                (j, k, C, t₂) ← m₂, (A ⟶ B , C) ← proj₁ Rs }
```

The definition mimics the standard definition of multiplication of matrices over real numbers. Namely, in the product `m₁ * m₂`, the parse trees from nonterminal `A` of the substring between positions `i` and `j` are composed from all parse trees from nonterminal `B` of substrings between `i` and `k` and from nonterminal `C` of substrings between `k` and `j` such that the rule `A ⟶ B , C` is in the grammar.

Each string `s` gives rise to an initial matrix. The initial matrix for `s` contains all the trees of type `s [ i, suc i )▶ A`. Recall that, to construct a tree of that type, we must use the second constructor of the definition of parse trees, which says that there must be a rule `A ⟶ t` in the grammar and the terminal `t` must be located at the `i`-th position of `s`. We denote the initial matrix by `m-init s`.

Next, we say that, to "raise" `m` to the `n`-th "power", we must compute the product of `m` raised to `k` and `m` raised to `n - k` for every nonzero `k` so that `k < n`.

```
pow : {s : String} → Mtrx s → ℕ → Mtrx s
pow {s} 0 m = if nullable
           then { (i, i, S, empt i) | i ← [0 ... length s) }
           else []
pow 1 m = m
pow n m = { t | k ← [1 ... n), t ← pow k m * pow (n - k) m }
```

The intuition for this function is that `pow s n` computes all the parse trees of substrings of `s` of length `n`. For substrings longer than one, we choose the position of a split `k` from the range `[1 ... n)`, then parse the left and right parts of the substring recursively, and finally assemble the subtrees by multiplying the corresponding matrices.

Finally, we can define an interface function for CYK parsing by rising the initial matrix to the power of `length s`.

```
cyk-parse : (s : String) → Mtrx s
cyk-parse s = pow (m-init s) (length s)
```

3.3 Correctness

The algorithm is correct if it assigns to a string the same parse trees as the parsing relation. We break the correctness statement down into soundness and completeness.

Soundness in the sense that `pow (m-init s) n` produces good parse trees of substrings of `s` is immediate by typing. With minimal reasoning we can also conclude that the `n`-th power produces strings of length `n`:

```
sound : (s : String) → (i j n : ℕ) → (A : N)
  → (t : s [ i, j )▶ A) → (i, j, A, t) ∈ pow (m-init s) n
  → j ≡ n + i
```

Next, we must show that, if some substring of `s` of length `n` is in the grammar, then it will definitely appear in the result of `pow (m-init s) n`:

```
complete : (s : String) → (i n : ℕ) → (A : N)
  → (t : s [ i, n + i )▶ A)
  → (i, n + i, A, t) ∈ pow (m-init s) n
```

Due to the simple recursive structure of parsing function, the proofs of soundness and completeness are also rather straightforward. Both theorems are proved by induction on the parse tree.

3.4 Termination

Another important aspect is termination. For the logic of Agda to be consistent, all functions must be terminating. This is statically checked by Agda's termination checker. So it is the duty of a programmer to provide sufficiently convincing arguments. This section describes the classical approach for proving termination based on well-founded relations [37].

The definition of `pow` given above makes two recursive calls to itself with arguments `k` and `n - k` where `k` comes from the range `[1 ... n)`. Morally, we understand that the function is terminating, but to convince Agda's type checker we have to explain that we make recursive calls along a well-founded relation.

Classically, we can say that a relation is well-founded, if it contains no infinite descending chains. An adequate constructive version uses the notion of accessibility.

An element `x` of a set `X` is called *accessible* with respect to some relation `_≺_`, if all elements related to `x` are accessible. Crucially, this definition is to be read inductively.

```
data Acc {X : Set}(_≺_: X → X → Set)(x : X) : Set where
  acc : ((y : X) → y ≺ x → Acc _≺_ y) → Acc _≺_ x
```

A relation can be said to be *well-founded*, if all elements in the carrier set are accessible.

```
Well-founded : {X : Set}(_≺_: X → X → Set) → Set
Well-founded = (x : X) → Acc _≺_ x
```

And we can prove that relation `_<_` on natural numbers is well-founded.

```
<-wf : Well-founded _<_
```

Finally, we must update the definition of function the `pow`. First, we observe that, for the function call `pow m n`, the recursive calls have shapes `pow n m k` and `pow m (n - k)`, where `k` comes from the interval `[1...n)`. Therefore, we prove following two lemmas:

```
<-lemma1 : (k : ℕ) → k ∈ [1 ··· n) → k < n

<-lemma2 : (k : ℕ) → k ∈ [1 ··· n) → n - k < n
```

By adding the accessibility proof as an additional argument and using it with `<-lemma1` and `<-lemma2` at the recursive calls, we discharge the obligations of the termination checker.

3.5 Memoization

Without memoization our implementation involves excessive recomputation of the matrices `pow m n`. To avoid recomputation of intermediate results, we implement a memoized version of the `pow` function.

We introduce a type of memo tables. A memo table can record some powers of `m` as entries; we allow only valid entries.

```
MemTbl : {s : String} → Mtrx s → Set
MemTbl {s} m = (n : ℕ) → Maybe (∃[m' : Mtrx s] m' ≡ pow m n)
```

In our implementation, extracting elements from the memo table takes time proportional to the number of elements in it. (Imperative implementations could do that in constant time.)

We introduce a function `pow-tbl` that is like `pow`, except that it expects to get some element `tbl` of `MemTbl m` as an argument. Instead of making recursive calls, it looks up matrices in the given memo table `tbl`. If the required matrix is not there, it falls back to `pow`. At this stage we do not worry about where to get a memo table from; we just assume that we have one given.

```
pow-tbl : {s : String} → (m : Mtrx s)
                       → ℕ → MemTbl m → Mtrx s
pow-tbl s m n tbl = if n < 2 then mt n
                             else { t | k ← [1 ... n),
                                        t ← mt k * mt (n - k) }
  where
    mt n = maybe (pow m n) fst (tbl n)
```

The next step is to prove that `pow` and `pow-tbl` compute propositionally equal results.

```
pow≡pow-tbl : {s : String} → (m : Mtrx s) → (n : ℕ)
    → (tbl : MemTbl m) → pow-tbl m n tbl ≡ pow m n
```

Now we have to find a way to actually build memo tables with intermediate results together with the proofs that they coincide with the matrices returned by `pow`. We implement a function which iteratively computes the powers `pow m n` of an argument matrix `m`, where `i` $\leq$ `n` $\leq$ `i + j` for given `i` and `j`, remembering all intermediate results.

```
pow-mem : {s : String} → (m : Mtrx s) → ℕ
                       → ℕ → MemTbl m → Mtrx s
pow-mem m i zero    tbl = pow-tbl m i tbl
pow-mem m i (suc j) tbl = pow-mem m (suc i) j tbl'
  where
    tbl' p = if p == i
             then just (pow-tbl m i tbl, pow≡pow-tbl m i tbl)
             else tbl p)
```

The function `pow-mem` calls itself with ever more filled memo tables starting from lower powers. Observe how the theorem `pow≡pow-tbl` is now used to ensure the correctness of each new memo table `tbl'`.

Finally, the memoized function for CYK parsing can be defined as follows:

```
cyk-parse-mem : (s : String) → Mtrx s
cyk-parse-mem s =
  pow-mem (m-init s) 0 (length s) (λ _ → nothing)
```

3.6 Example

In this section, we illustrate some aspects of the functions developed. For the sake of an example, we define a recognizer function that counts the number of derivations for a given string.

```
cyk-recognizer : String → ℕ
cyk-recognizer s = length (cyk-parse-mem s)
```

We use the grammar that was defined in the previous section, but package it into a record `G`:

```
G : GrammarCNF
G = record {
   Rule-Desc = Rule-Desc ;
   S = ⟨'A'⟩ ;
   nullable = false ;

   RHS-prop = RHS-prop ;
   S-prop = S-prop
 }
```

The language of `G` (i.e., derivations from the start nonterminal) consists of strings of terminal `1` of length greater than or equal to 2. The strings `1111` and `11101` have respectively one and zero derivations:

```
parse1 : cyk-recognizer G [ ⟨'1'⟩, ⟨'1'⟩, ⟨'1'⟩, ⟨'1'⟩ ] ≡ 1
parse1 = refl

parse2 : cyk-recognizer G [ ⟨'1'⟩, ⟨'1'⟩, ⟨'0'⟩, ⟨'1'⟩ ] ≡ 0
parse2 = refl
```

Note that the first argument supplied to `cyk-recognizer` is the grammar `G`. This is because all of the development in this section is parameterized by a particular CFG.

Alternatively, we can define a CFG `G'` with the following rules:

```
Rule-Desc' = fsd-nodup [  ⟨'A'⟩ ⟶ [ nt ⟨'B'⟩, nt ⟨'B'⟩ ]
                       ,  ⟨'B'⟩ ⟶ [ nt ⟨'B'⟩, nt ⟨'B'⟩ ]
                       ,  ⟨'B'⟩ ⟶ [ tm ⟨'1'⟩ ] ]
```

The languages of `G` and `G'` are the same. However, `G'` is an ambiguous grammar while `G` is unambiguous. The consequence is that the strings in the language of `G'` have exponentially many derivations:

```
parse3 : cyk-recognizer G' [ ⟨'1'⟩, ⟨'1'⟩, ⟨'1'⟩, ⟨'1'⟩ ] ≡ 5
parse3 = refl
```

3.7 Conclusions

In this section, we implemented the CYK parsing algorithm for context-free grammars. The algorithm was implemented by correctness-preserving refinement from the naive version of the algorithm which is inefficient but for which the correctness is easily provable. The refinement strategy helps in gradual introduction of performance-related aspects of the algorithm and modularizing the overall correctness proof.

The CYK algorithm operates on grammars in Chomsky normal form. To be able to use our code on arbitrary context-free grammars, in the next section we implement normalization of grammars.

4 Normalization of context-free grammars

In this section (summarizing Paper III), we prove in Agda the classical theorem that every context-free grammar can be transformed into an equivalent one (i.e., giving rise to the same language) in Chomsky normal form. This is accomplished by a sequence of four transformations.

We prove formally that each of these transformations is correct in the sense of making progress toward normality and preserving the language of the given grammar. Also, we show that the right sequence of these transformations leads to a grammar that is indeed in Chomsky normal form, since each next transformation preserves the normality properties established by the previous ones, and accepts the same language as the original grammar. In our constructive setting, we can arrange our formalization so that soundness and completeness proofs are functions converting between parse trees in the normalized and original grammars.

The full normalization transformation for a CFG is the composition of the following constituent transformations [26]:

1. elimination of all *ε-rules* (i.e., rules of the form $\mathtt{A} \longrightarrow \varepsilon$);
2. elimination all *unit rules* (i.e., rules of the form $\mathtt{A} \longrightarrow \mathtt{B}$);
3. replacing all rules $\mathtt{A} \longrightarrow \mathtt{X}_1\mathtt{X}_2 \ldots \mathtt{X}_k$ where $\mathtt{k} \geq 3$ with rules $\mathtt{A} \longrightarrow \mathtt{X}_1\mathtt{A}_1$, $\mathtt{A}_1 \longrightarrow \mathtt{X}_2\mathtt{A}_2$, $\mathtt{A}_{k\text{-}2} \longrightarrow \mathtt{X}_{k\text{-}1}\mathtt{X}_k$ where $\mathtt{A}_i$ are "fresh" nonterminals;
4. for each terminal a, adding a new rule $\mathtt{A} \longrightarrow \mathtt{a}$ where A is a fresh nonterminal and replacing a in the right-hand sides of all rules with length at least two with A.

In Section 4.1, we give a definition of the parsing relation for grammars in general form. In Section 4.2, we describe the elimination of unit rules and the correctness properties of this transformation. The other three transformations are defined and proved correct similarly. In Section 4.3, we explain how to achieve full normalization after implementing and proving the correctness properties of all four transformations by appropriately chaining them together and establishing that every next transformation preserves the normality aspects achieved by the previous ones. Finally, in Section 4.4, we connect the normalization of a grammar and CYK parsing to achieve correct fully general context-free parsing.

The Agda development corresponding to the content of this section is located in the modules `CNF` and `GeneralParsing` of the accompanying code.

4.1 Parsing relation

In the previous section, we defined the type `GrammarCNF` for grammars in Chomsky normal form and the parsing relation tuned for CYK algorithm and CNF gram-

mars. Now, we define grammars and the parsing relation for grammars in general form:

```
record Grammar : Set₁ where
  field
    Rule-Desc : FinSubDesc RB _=RB?_ true
    S : N

  Rule : Set
  Rule = Elem Rule-Desc

  Rs : Listable Rule
  Rs = lstblElem Rule-Desc

  _=R?_ : DecEq Rule
  _=R?_ = deqElem RuleDesc

  projL : Rule → N
  projL (r , _) = proj1 r

  projR : Rule → RHS
  projR (r , _) = proj2 r
```

The type `Grammar` differs from `GrammarCNF` by not having the nullability flag, and by the restrictions on the start nonterminal and on the right hand sides of the productions. As in previous section, we assume types `N` and `T` for nonterminals and terminals together with decidable equality on their elements. Decidable equalities on `N` and `T` give rise to decidable equality on `N × List (N ⊎ T)` denoted by `_=RB?_`. We also denote a rule `r : Rule` as `A ⟶ rhs` if $\texttt{proj}_{\text{L}}$ `r ≡ A` and $\texttt{proj}_{\text{R}}$ `r ≡ rhs`.

Since some of the passes of normalization require generation of fresh nonterminals, it is better to have `N` infinite. For more details on that matter, we refer to Section 5 of Paper III.

The datatype of parse trees (abstract syntax trees) is parameterized by a grammar `G` and indexed by a root nonterminal and a string (list of terminals to parse):

```
mutual
  Tree (G : Grammar) : N → String → Set

  Forest (G : Grammar) : RHS → String → Set
```

In general, the type `Tree G A s` collects all parse trees for a string `s` with respect to a grammar `G` and a nonterminal `A` at the root. The auxiliary type `Forest G rhs s` collects all parse forests for a string `s` whose constituent individual parse trees are rooted at the symbols in `rhs`.

To construct a tree of type `Tree G A s`, we must choose a rule `A ⟶ rhs` of the type `Rule` from `G` and provide a forest `Forest G rhs s` of trees starting from the nonterminals of `rhs`. The terminals of `rhs` are treated as leaves and the concatenation of all leaves of the tree must result in the string `s`.

In our original Paper III, the grammar was modelled simply by a list of rules `Rs`. Therefore, the constructor of `Tree` took a rule `r` together with a proof that `r ∈ Rs`. This made the resulting parse trees have proofs of list membership to appear at every node. Additionally, in such a solution parse trees become invalid, when the original list of rules changes slightly. Now, we avoid the problems of dealing with lists and explicit positions in lists by using listable sets defined by descriptions of finite subsets.

4.2 Unit rule elimination

A single step of unit rule elimination is made by the function `nu-step`:

```
nu-step : Grammar → N → Grammar
```

Compared to the grammar `G`, in the grammar `nu-step G A`, every rule of the form `B ⟶ [ nt A ]` is replaced with all rules of the form `B ⟶ rhs'` where `rhs'` stands for a right-hand side such that `A ⟶ rhs'` is in `G` and `rhs' ≢ [ nt A ]`. The computation is based on the underlying descriptions `Rule-Desc G` of the rule set.

Now, full unit rule elimination is achieved by applying this procedure to all nonterminals:

```
norm-u : Grammar → Grammar
norm-u G = foldl nu-step G (NTs G)
```

The function `NTs` returns an enumeration of all nonterminals used in a given grammar.

The correctness of this transformation can be divided into progress, soundness and completeness.

Progress First, it must be shown that `nu-step` achieves some progress towards normality of the grammar:

```
nu-step-progress : (G : Grammar) → (B : N)
   → (r : Rule (nu-step G B)) → proj_R r ≢ [ nt B ]
```

This lemma states that there is no rule with the right-hand side `[ nt B ]` in the grammar `nu-step G B`. The progress lemma for the `norm-u` is a trivial consequence:

```
nu-progress :  (G : Grammar) →  (r : Rule (norm-u G))
   → (A : N) → proj_R r ≢ [ nt A ]
```

Soundness Second, `nu-step` is sound, namely, any parse tree of a string `s` in the transformed grammar should be parsable in the original grammar.

```
nu-step-sound : (G : Grammar) → (A B : N) → (s : String)
                    → Tree (nu-step G B) A s → Tree G A s
```

The straightforward lifting of this lemma gives soundness of `norm-u`

```
nu-sound : (G : Grammar) → (A : N) → (s : String)
                    → Tree (norm-u G) A s → Tree G A s
```

Completeness Third, any string parsable in the original grammar is parsable in the transformed one.

```
nu-step-complete : (G : Grammar) → (A B : N) → (s : String)
                    → Tree G A s → Tree (nu-step G B) A s
```

This property induces completeness of `norm-u`:

```
nu-complete : (G : Grammar) → (A : N) → (s : String)
                    → Tree G A s → Tree (norm-u G) A s
```

4.3 Full normalization and correctness

As we discussed previously, full normalization consists of four separate grammar transformations suitably chained together. Unit rule elimination (`norm-u`) was described above. Epsilon rule elimination (denoted by `norm-e`) removes rules with an empty right-hand side. The transformation `norm-l` transforms a grammar so that the right-hand sides of all rules contain at most two symbols. The transformation `norm-t` modifies a grammar so that the right-hand sides longer that one symbol do not contain terminals.

The correctness properties proved for `norm-e`, `norm-t`, and `norm-l` are similar to the correctness properties of `norm-u`. However, there are some subtleties. The transformation `norm-e` removes all epsilon rules, therefore, the empty word cannot belong to the language of the transformed grammar. So, completeness of the `norm-e` transformation holds only for non-empty strings.

```
ne-complete : (G : Grammar) → (A : N) → (s : String)
           → s ≠ [] → Tree Rs A s → Tree (norm-e G) A s
```

In the next section, we describe how the restriction on `s` can be removed for the start nonterminal by converting grammars to the type `GrammarCNF` which has the nullability flag.

Another subtlety is that, to achieve their goals, the transformations `norm-l` and `norm-t` may need to introduce fresh nonterminals into the result grammar. This means that soundness for these transformations holds only for trees that are rooted by nonterminals from the original grammar.

```
nl-sound : (G : Grammar) → (A : N) → (s : String)
   → A ∈ NTs G → Tree (norm-l G) A s → Tree G A s
```

Finally, the full normalization function is defined simply by composition:

```
norm : Grammar → Grammar
norm = norm-u ∘ norm-e ∘ norm-t ∘ norm-l
```

The function `norm` is the composition of the four transformations we have introduced. The order in which these transformations are chained matters. For example, `norm-e` can add new unit rules, so `norm-u` must be performed after `norm-e`.

Progress To prove progress of `norm`, we must establish that each transformation preserves the progress made by previous ones. For example, it can be easily seen that, since `norm-e` never introduces new right-hand sides to the transformed grammar, it is clear that the progress achieved by `norm-t` and `norm-l` is preserved.

After having formalized the preservation of progress for all transformations, we prove that the grammar `norm G` is in Chomsky normal form.

```
norm-progress : (G : Grammar) → (r : Rule (norm G))
    → (∃[ B C ∈ N ] proj_R r ≡ [ nt B, nt C ]) ⊎
       (∃[ a ∈ T ] proj_R r ≡ [ tm a ])
```

Soundness To show soundness of `norm`, we only need to chain the soundness results of the individual transformations:

```
norm-snd : (G : Grammar) → (s : String) → (A : N)
  → A ∈ NTs G → Tree (norm G) A s → Tree G A s
```

Completeness As in the case of soundness, completeness of `norm` is proved by chaining the completeness results of the constituent transformations:

```
norm-cmplt : (G : Grammar) → (s : String) → (A : N)
  → s ≢ [] → Tree G A s → Tree (norm G) A s
```

4.4 General context-free parsing

Now, we can implement a function that converts any general context-free grammar to a grammar in Chomsky normal form.

```
toCNF : Grammar → GrammarCNF
```

The conversion simply uses the function `norm` on its argument and then `norm-progress` establishes a property of the right-hand sides. Also, if the start nonterminal appears in the right-hand sides of the rules of `norm G`, then a fresh start nonterminal `S'` must be created and, for every rule `S G ⟶ rhs`, the rule `S' ⟶ rhs` must be added to the resulting CNF grammar. This will guarantee

that the fresh nonterminal `S'` never appears on the right. CNF grammars additionally have a Boolean field `nullable` that indicates whether the empty word is in the language or not. Therefore, the nullability flag is set accordingly to whether the start nonterminal is nullable.

The language of a grammar is a collection of all strings parsable from the start nonterminal:

```
TreeS : Grammar → String → Set
TreeS G s = Tree (Rs G) (S G) s

TreeSCNF : GrammarCNF → String → Set
TreeSCNF G s = let [_,_)▶_ = [_,_)▶_ G in
                   [ 0 , length s )▶ (S G)
```

(The definition of `TreeSCNF` must deal with the fact that parse trees for CNF grammars are parameterized by a global grammar `G`. Therefore, we must supply it as a first argument.)

We can now show that the languages of the original and normalized grammars are same:

```
complete : (s : String) → TreeS G s → TreeSCNF (toCNF G) s

sound : (s: String) →  TreeSCNF (toCNF G) s → TreeS G s
```

Finally, we can implement the function for general context-free grammar parsing:

```
cyk-parse-gen : (G : Grammar) → (s : String)
                              → List (TreeS G s)
```

General context-free parsing is done in three steps. First, the original grammar `G` is normalized into a grammar `G'`. Next, the CYK algorithm is used to get a list of parse trees of `s`. Finally, the trees from start nonterminals of `G'` in normalized grammar are converted to the original grammar by using the function `sound`.

The final correctness property is

```
cyk-parse-gen-correct : (G : Grammar) → (s : String)
  → (t : TreeS G s)
  → ∃[ t' ∈ TreeS G s ] t' ∈ cyk-parse-gen G s
```

We must remark that the number of parse trees for a string `s` of a general grammar `G` can be infinite. Therefore, given a tree `t : TreeS G s`, we cannot always imply `t ∈ cyk-parse-gen G s`. Therefore, the correctness proof states only that, if string `s` is in the language of the grammar, then we can construct some parse trees of `s`. In the future work section, we discuss that there is a close correspondence between the trees `t` and `t'`, namely, the tree `t'` is a "normalized" version of the tree `t`.

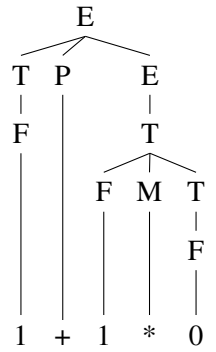
4.5 Example

Let us define the following unambiguous context-free grammar for arithmetic expressions made up of addition, multiplication and digits 0 and 1:

```
_⟶_ : {A B : Set} → A → B → A × B
_⟶_ = _,_

G : Grammar
G = record {
   S = "E" ;
   Rule-Desc = fsd-plain rules
 }
  where
    rules =  [ "E" ⟶  [ nt "T" , nt "P" , nt "E" ]
             , "E" ⟶  [ nt "T" ]

             , "T"  ⟶ [ nt "F" , nt "M" , nt "T" ]
             , "T"  ⟶ [ nt "F" ]

             , "F"  ⟶ [ tm '0' ]
             , "F"  ⟶ [ tm '1' ]

             , "P" ⟶ [ tm '+' ]
             , "M" ⟶ [ tm '*' ] ]
```

(The `_⟶_` is a synonym for product constructor.) By using the unit rules, the grammar `G` encodes the right associativity of addition and multiplication, and the precedence of multiplication over addition.

Now, when we execute the function `cyk-parse-gen G` on the string `"1+1*0"`, we get the following parse tree:

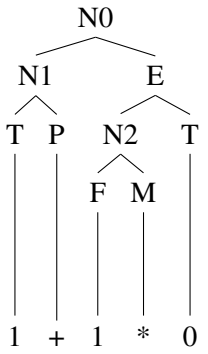


However, let us look at how this result is computed. In the beginning, the grammar `G` is normalized:

```
normS G = record{
   S = "N0" ;
   Rule-Desc = fsd-plain rules'
}
  where
   rules' = [ "N0" ⟶ [ nt "N1" , nt "E" ]
            , "N0" ⟶ [ nt "N2" , nt "T" ]

            , "N1" ⟶ [ nt "T" , nt "P" ]
            , "N2" ⟶ [ nt "F" , nt "M" ]

            , "E" ⟶ [ nt "N1" , nt "E" ]
            , "E" ⟶ [ nt "N2" , nt "T" ]
            , "T" ⟶ [ nt "N2" , nt "T" ]

            , "E" ⟶ [ tm '0' ]
            , "E" ⟶ [ tm '1' ]
            , "F" ⟶ [ tm '0' ]
            , "F" ⟶ [ tm '1' ]
            , "T" ⟶ [ tm '0' ]
            , "T" ⟶ [ tm '1' ]

            , "P" ⟶ [ tm '+' ]
            , "M" ⟶ [ tm '*' ] ]
```

It can be seen easily that the grammar `normS G` is in normal form. There are three fresh nonterminals – `N0`, `N1` and `N2`. The start nonterminal changed, because the previous one `E` appeared in the right-hand sides of some rules.

Next, the CYK algorithm uses the normalized grammar to perform parsing. The parse tree of the string `"1+1*0"` for the grammar `normS G` looks as follows:



By looking at that tree we can figure out the reason why each new rule was introduced into the normalized grammar. The function `sound` "undoes" the effect of

normalization to get a parse tree for the grammar `G`.

4.6 Conclusions

The goal of this section was to implement a certified normalization procedure for context-free grammars. We started by defining the general parsing relation and a record type for context-free grammars with a distinguished start nonterminal. Then we explained that the normalization transformation can be decomposed into four simpler transformations. Next we described unit rule elimination and explained that each transformation must preserve the language of the original grammar (soundness and completeness proofs), achieve the progress towards normalization and preserve the progress of earlier transformations.

Since the proofs are done constructively, one can execute an instance of the soundness proof to convert a parse tree for transformed grammar to a parse tree for the original grammar. This is a situation typical of Agda formalizations: we prove a theorem and its proof also serves as the function that we are really interested in from the programming perspective.

Finally, we combined CYK parsing and normalization of grammars into a certified parsing algorithm for general context-free grammars.

5 Related work

In this section, we continue to discuss the related work. We also give some account of the papers that appeared subsequent to our publications. Some of those also mention our work.

Many attempts were made to formalize the theory of formal languages. We start by describing some of the earliest we could find. Kreitz [30] gave a formalization of pumping lemma for the theory of finite automata in the Nuprl system. Courant and Filliâtre worked on the theory of regular and context-free languages in Coq [16, 22]. Among other results, they showed that all context-free languages can be recognized by pushdown automata and that context-free languages are closed under the union. Constable et al. [14] formalized the theory of regular languages in Nuprl and proved the Myhill–Nerode theorem. It is worth noticing that all of the three above mentioned developments were carried out in the setting of constructive type theory. Nipkow [36] provided a verified lexical analyzer implemented in Isabelle/HOL. Isabelle/HOL is not constructive, but the author took care to ensure that most of the definitions are executable.

The interest toward formalization of parsing started to grow after some substantial work was done in the field of certified compilation. The paper [32] by Leroy discussed the full verification of compilation of a subset of C language using Coq. Strecker [48] also reported on compiler verification for C0. However, the verified phases of a compiler did not include lexing and parsing. Thereby, some demand was created for formalization and verification of parsing algorithms. Barthwal and Norrish [9, 6] presented a verified SLR parser generator for CFGs that is able to handle a subset of unambiguous grammars. Jourdan et al. [27] focused on LR grammars which is a wider class than SLR grammars. Danielsson and Norell [17] implemented a library of parser combinators in Agda for grammars without left recursion.

None of the previously mentioned formalized developments can parse general context-free languages. Earley [21] gave a description of an efficient general parsing algorithm for CFGs. It was shown that Earley's algorithm has the same worst-case time complexity as the CYK approach, but outperforms it on some classes of context-free grammars. The correspondence between Earley's and the CYK algorithms was properly analyzed by Graham and Harrison [24]. Ridge [44] used techniques from Earley's parsing and produced a generic parser generator with proofs of correctness in HOL4. However, the time complexity of the memoized version of the implemented parser was estimated to be $O(n^5)$. In his latest work [45], Ridge showed that the performance issue can be fixed and devised a fully verified parser that also has good practical performance.

Barthwal and Norrish [9, 6] formalized SLR parsing using the HOL4 proof assistant (SLR grammars are a subset of $LR(1)$ grammars). They constructed an SLR parser for context-free grammars, and proved it to be sound and complete.

The formalization of the SLR parser was done in over 20 000 lines of code. Our implementation of the CYK parser is less than 2000 loc.

Also, Barthwal and Norrish [7, 6] described a formalization of the Chomsky and Greibach normal forms for context-free grammars with the HOL4 theorem prover. They showed how to solve the problems which arise from mechanizing the straightforward pen and paper proofs. The non-constructive setting gave the advantage of the power of extensional and classical reasoning, but also the significant drawback that it did not deliver actual functions for normalizing grammars or converting parse trees between grammars. More precisely, the transformations were described in a relational style. Also, the authors mentioned that they did not want to work constructively, as this had the advantage of reasoning extensionally about sets. The formalization of normal forms (Chomsky and Greibach) by Barthwal and Norrish were done in about 14000 lines of code with 700 lemmas and theorems. For comparison, our formalization (CNF only) is less than 4800 loc.

Recently, Barthwal and Norrish [8] continued formalization of general context-free language theory in HOL4. They proved that pushdown automata and context-free grammars accept the same languages and used this result to show some closure properties of context-free languages such as union and concatenation.

Valiant's algorithm [50] is an extension of the CYK algorithm and computes the same parsing table as the CYK algorithm. However, Valiant showed that the parsing problem is reducible to Boolean matrix multiplication. A formalization of Valiant's algorithm in Agda was originally carried out by Sjöblom [47]. Later, Bernardy and Jansson [11] reimplemented Valiant's algorithm for context-free parsing in Agda. They presented an algebraic specification, an implementation, and a proof of correctness. Their generalization of the algorithm can be used for calculation of the multiplicative transitive closure of upper triangular matrices.

Bernardy and Claessen [10] showed that the divide and conquer structure of Valiant's parsing algorithm yields an efficient parallel algorithm. In particular, if the input is hierarchic, then the conquer step can be computed faster than is required by matrix multiplication.

Boolean grammars are an extension of context-free grammars, in which the rules may contain conjunction and negation of several right-hand sides. Conjunction and negation are used to specify the intersection and complement of languages. Okhotin [40] showed how to extend Valiant's algorithm to parse Boolean grammars.

Ramos and de Queiroz have performed a number of formalizations of theory of context-free languages in Coq. The earliest paper [41] reported on the proofs of the correctness of the concatenation, union and closure operations. They implemented [42] useless symbol elimination, inaccessible symbol elimination, unit rules elimination and empty rules elimination operations and proved the implementation correct in the sense of preservation of the language generated by the original grammar. In their most recent work, Ramos et al. [43] extended their

development and proved the existence of Chomsky normal form for context-free grammars and the pumping lemma for context-free languages.

The formalization of theory of regular languages is also actively developing. In Paper V (the candidate's master thesis work), we reported on a certified parser generator for regular languages using matrix combinators. Braibant and Pous [13] developed a tactic for deciding Kleene algebras in Coq. Coquand and Siles [15] used Coq to develop a decision procedure for equivalence of regular languages defined by regular expressions.

Moreira et al. [5, 33] implemented an algorithm in Coq which decides regular expression equivalence through an iterated process of testing the equivalence of their partial derivatives.

Doczkal et al. [19] presented a concise formalization of regular languages in Coq. Among other results they gave a minimization algorithm for DFAs and prove its uniqueness. Also they implemented functions that translate regular expressions to DFAs and NFAs and proved correctness of these transformations. Later, Doczkal and Smolka [20] also verified translations from two-way automata to one-way automata. The translation uses a constructive variant of the Myhill-Nerode theorem. The authors noted that their formalization makes extensive use of countable and finite types provided by Coq/Ssreflect.

Korkut et al. [29] implemented Harper's matching algorithm [25] for regular expressions. They illustrated how defunctionalization of the matcher allows Agda to see termination without an explicit metric.

6 Conclusions

In this section, we summarize the main points of the thesis and also discuss the possible directions for future work.

6.1 Summary

In this thesis, we concentrated on formalizing the theory of context-free languages. We addressed the aspects related to definition of grammars, parsing and normalization. To define a context-free grammar one needs to be able to specify finite sets. We studied listability of sets in the constructive setting and implemented viable solutions to boilerplate-free programming with listable sets. One insight there was that decidable predicates can be squashed losslessly. Based on this idea, we devised an approach for defining new listable sets as subsets of a base set by providing a list of elements of that set. Another important insight is that listability of a set implies decidable equality. This allowed us to implement a toolbox of combinators with the necessary preconditions checked automatically during the type-checking phase.

Next, we implemented the CYK parsing algorithm for context-free grammars in Chomsky normal form. Thanks to the simplicity of the algorithm and its restriction to normalized grammars, we were able to prove the correctness of our implementation relatively straightforwardly. Moreover, we started with the functional, non-memoized version, which also eased our proofs. Then we defined certified memoization tables and developed a memoized version of algorithm based on these tables with correctness preservation guarantees.

Finally, we implemented a certified normalization procedure for context-free grammars. We followed the classical approach of defining normalization as the composition of some small transformations in a certain order. For each transformation, we proved that it preserves the language of a grammar, achieves progress towards normality, and preserves the progress made by previous transformations. To prove preservation of the language, we proved soundness and completeness of the transformation. Soundness tells us that any word in the language of the transformed grammar is also in the language of the original grammar. Completeness goes in the opposite direction. Crucially, these proofs are constructive, so one can execute an instance of the soundness proof to convert a parse tree for transformed grammar to the parse tree for the original grammar.

When packaged together, this toolset allows one to concisely define a context-free grammar, normalize it, perform CYK parsing and transform the resulting parse trees into parse trees for original grammar.

6.2 Future work

An important property of normalized context-free grammars is that every string has at most a finite number of derivations. General context-free grammars do

not have that property, because of the unit and epsilon rules. In some cases, the unit rules can be arranged in cycles and the empty word can have infinitely many derivations. We believe that parse trees that differ only in these two aspects could be treated as morally the same derivations and the smallest tree is a normal form. We would like to prove that the conversion of a parse tree for the normalized grammar to the original grammar returns normal trees. Then normalization of parse trees by passing through the normalized grammar can be seen as a form of normalization-by-evaluation.

An attribute grammar is a context-free grammar that has been extended to provide context-sensitive information by attaching attributes to some of its nonterminals. Computationally, an attribute grammar is a specification for attribute evaluation. However, it is easy to define grammars so that the dependency between attributes will be cyclic, causing lazy demand-driven attribute evaluation to diverge. An attribute grammar is called *cyclic*, if it is viable to construct a parse tree where an attribute of a particular node depends on itself. In the seminal paper on attribute grammars [28], Knuth gave an algorithm for deciding whether an attribute grammar is cyclic or not. It seems possible to implement this algorithm in Agda together with proofs of correctness (soundness and completeness). This certified implementation of the acyclicity check could be then used to define a provably terminating evaluator for acyclic attribute grammars. Our first steps in this direction [23] were made by describing the cyclic paths on trees and showing how cycles on trees can be decomposed into smaller cycles. Then we established that this decomposition implies an induction principle for cycles. By using this decomposition and induction principle, we plan to implement the acyclicity checker and prove that it is correct (sound and complete).

Another possible line for future research is disambiguation of parsing. An ambiguous context-free grammar defines a language where some strings have multiple derivations. In programming languages, the underlying CFGs are often ambiguous. A common example is the dangling "else" problem. In these cases, the ambiguities are generally resolved by adding precedence rules. It could be useful to formalize disambiguation as a transformation of a CFG according to the given set of precedence rules. Similarly to our work on normalization, soundness of transformation will become an (executable) function converting the parse trees from the transformed to the original grammar.

References

[1] The Agda Team. The Agda wiki, 2015. `http://wiki.portal.chalmers.se/agda/`

[2] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Pearson Education, 2006.

[3] A. V. Aho, J. D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice Hall, 1972.

[4] J. B. Almeida, M. J. Frade, J. S. Pinto, S. M. de Sousa. *Rigorous Software Development.* Springer, 2011.

[5] J. B. Almeida, N. Moreira, D. Pereira, S. M. de Sousa. Partial derivative automata formalized in Coq. In M. Domaratzki, K. Salomaa, eds., *Revised Selected Papers from 15th Int. Conf. on Implementation and Application of Automata, CIAA 2010*, v. 6482 of *Lect. Notes in Comput. Sci.*, pp. 59-68. Springer, 2011.

[6] A. Barthwal. *A Formalisation of the Theory of Context-Free Languages in Higher Order Logic.* PhD thesis, Australian National University, Canberra, 2010.

[7] A. Barthwal, M. Norrish. A formalisation of the normal forms of context-free grammars in HOL4. In A. Dawar and H. Veith, eds., *Proc. of 24th Workshop on Computer Science Logic, CSL 2010*, v. 6247 of *Lect. Notes in Comput. Sci.*, pp. 95–109. Springer, 2010.

[8] A. Barthwal, M. Norrish. A mechanisation of some context-free language theory in HOL4. *J. of Comput. and Syst. Sci.*, v. 80(2), pp. 346–362, 2014.

[9] A. Barthwal, M. Norrish. Verified, executable parsing. In G. Castagna ed., *Proc. of 18th Europ. Symp. on Programming Languages and Systems, ESOP '09*, v. 5502 of *Lect. Notes in Comput. Sci.*, pp. 160–174. Springer, 2009.

[10] J.-P. Bernardy, K. Claessen. Efficient parallel and incremental parsing of practical context-free languages. *J. of Funct. Prog.*, v. 25, article e10, 2015.

[11] J.-P. Bernardy, P. Jansson. Certified context-free parsing: a formalisation of Valiant's algorithm in Agda. *Log. Meth. in Comput. Sci.*, v. 12(2), article 6, 2016.

[12] A. Bove, P. Dybjer, U. Norell. A brief overview of Agda, a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, M. Wenzel, eds., *Proc. of 22nd Int. Conf. on Theorem Proving in Higher Order*

Logics, TPHOLs 2009, v. 5674 of *Lect. Notes in Comput. Sci.*, pp. 73–78. Springer, 2009.

[13] T. Braibant, D. Pous. Deciding Kleene algebras in Coq. *Log. Meth. in Comput. Sci.*, v. 8(1), article 16, 2012.

[14] R. L. Constable, P. B. Jackson, P. Naumov, J. Uribe. Constructively formalizing automata. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 213–238. MIT Press, 1998.

[15] T. Coquand, V. Siles. A decision procedure for regular expression equivalence in type theory. In J. P. Jouannaud, Z. Shao, eds., *Proc. of 1st Int. Conf. on Certified Programs and Proofs, CPP 2011*, v. 7086 of *Lect. Notes in Comput. Sci.*, pp. 119–134. Springer, 2011.

[16] J. Courant, J. Filliâtre. Beginning of formal language theory, 1993. Available from `http://www.lix.polytechnique.fr/coq/V8.2pl1/contribs/Automata.html`

[17] N. A. Danielsson. Total parser combinators. In *Proc. of 15th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '10*, pp. 285–296, ACM, 2010.

[18] A. Deursen, P. Klint, J. Visser. Domain-specific languages: an annotated bibliography. In *ACM SIGPLAN Not.*, v. 35, pp. 26–36, ACM, 2000.

[19] C. Doczkal, J.-O. Kaiser, G. Smolka. A constructive theory of regular languages in Coq. In G. Gonthier, M. Norrish, eds., *Proc. of 3rd Int. Conf. on Certified Programs and Proofs, CPP 2013 (Melbourne, Dec. 2013)*, v. 8307 of *Lect. Notes in Comput. Sci.*, pp. 82–97. Springer, 2013.

[20] C. Doczkal, G. Smolka. Two-way automata in Coq. In J. Blanchette, S. Merz, eds., *Proc. of 7th Int. Conf. on Interactive Theorem Proving, ITP 2016*, v. 9807 of *Lect. Notes in Comput. Sci.*, pp. 151–166. Springer, 2016.

[21] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, v. 13(2), pp. 94–102, 1970.

[22] J.-C. Filliâtre. Finite automata theory in Coq: a constructive proof of Kleene's theorem. Technical Report 97-04, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 1997.

[23] D. Firsov, T. Uustalu. Acyclic attribute evaluation in a dependently typed setting. In *Abstracts of 27th Nordic Workshop on Programming Theory, NWPT 2015*, Technical report RUTR-SCS16001, School of Computer Science, pp. 124–126, Reykjavik University, 2016.

[24] S. L. Graham, M. A. Harrison. Parsing of general context-free languages. *Advances in Computing*, v. 14, pp. 77–185, 1976.

[25] R. Harper. Proof-directed debugging. In *J. of Funct. Program.*, v. 9(4), pp. 463–469, 1999. Corrigendum in v. 19(2), p. 262, 2009.

[26] J. E. Hopcroft, J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[27] J.-H. Jourdan, F. Pottier, X. Leroy. Validating LR(1) parsers. In H. Seidl ed., *Proc. of 21st Europ. Symp. on Programming, ESOP 2012*, v. 7211 of *Lect. Notes in Comput. Sci.*, pp. 397–416. Springer, 2012.

[28] D. E. Knuth. Semantics of context-free languages. In *Math. Syst. Theor.*, v. 2, pp. 127–148, 1968.

[29] J. Korkut, M. Trifunovski, D. R. Licata. Intrinsic verification of a regular expression matcher. Manuscript, 2016.

[30] C. Kreitz. Constructive automata theory implemented with the Nuprl proof development system. Technical Report TR 86-779, Dept. of Computer Science, Cornell University, 1986.

[31] K. Kuratowski. *Sur la notion d'ensemble fini*. *Fund. Math.* 1(1), pp. 129–131, 1920. Available at `https://eudml.org/doc/212596`

[32] X. Leroy. Formal verification of a realistic compiler. *Commun. of ACM*, v. 52(7), pp. 107–115, 2009.

[33] N. Moreira, D. Pereira, S. M. de Sousa. Kleene algebra terms equivalence in Coq. In *J. Log. Algebr. Meth. Program.*, v. 84(3), pp. 377–401, 2015.

[34] M. Müller-Olm, D. A. Schmidt, B. Steffen. Model checking: a tutorial introduction. In G. File and A. Cortesi, eds., *Proc. of 6th Static Analysis Symposium, SAS '99*, v. 1694 of *Lect. Notes in Comput. Sci.*, pp. 330–354. Springer, 1999.

[35] H. R. Nielson, F. Nielson. *Semantics with Applications: A Formal Introduction*, *Wiley Professional Computing*. John Wiley and Sons, 1992.

[36] T. Nipkow. Verified lexical analysis. In J. Grundy, M. Newey, eds., *Proc. of 11th Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLS '98*, v. 1479 of *Lect. Notes in Comput. Sci.*, pp. 1–15. Springer, 1998.

[37] B. Nordström. Terminating general recursion. *BIT Numerical Mathematics*, v. 28(3), pp. 605–619, 1988.

[38] U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, S. D. Swierstra, eds., *Revised Lectures from 6th Int. School on Advanced Functional Programming, AFP 2008*, v. 5832 of *Lect. Notes in Comput. Sci.*, pp. 230–266. Springer, 2009.

[39] U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory.* PhD thesis, Chalmers University of Technology, Göteborg, 2007.

[40] A. Okhotin. Parsing by matrix multiplication generalized to boolean grammars. *Theor. Comput. Sci.*, v. 516, pp. 101–120, 2014.

[41] M. V. M. Ramos, R. J. G. B. de Queiroz. Formalization of closure properties for context-free grammars. CoRR abs/1506.03428, 2014.

[42] M. V. M. Ramos, R. J. G. B. de Queiroz. Formalization of simplification for context-free grammars. CoRR abs/1509.02032, 2015.

[43] M. V. M. Ramos, R. J. G. B. de Queiroz, N. Moreira, J. B. Almeida. On the formalization of some results of context-free language theory. In J. Väänänen, Å. Hirvonen, R. de Queiroz, eds., *Proc. of Workshop on Logic, Language, Information and Computation, WoLLIC 2016*, v. 9803 of *Lect. Notes in Comput. Sci.*, pp. 338–357. Springer, 2016.

[44] T. Ridge. Simple, functional, sound and complete parsing for all context-free grammars. In J.-P. Jouannaud, Z. Shao, eds., *Proc. of 1st Int. Conf. on Certified Programs and Proofs, CPP 2011*, v. 7086 of *Lect. Notes in Comput. Sci.*, pp. 103–118. Springer, 2011.

[45] T. Ridge. Simple, efficient, sound and complete combinator parsing for all context-free grammars, using an oracle. In B. Combemale, D. J. Pearce, O. Barais, J. J. Vinju, eds., *Proc. of 7th Conf. on Software Language Engineering, SLE 2014*, v. 8706 of *Lect. Notes in Comput. Sci.*, pp. 261–281. Springer, 2014.

[46] M. Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 1996.

[47] T. B. Sjöblom. An Agda proof of the correctness of Valiant's algorithm for context free parsing. MSc thesis, Chalmers University of Techn., 2013.

[48] M. Strecker. Compiler verification for C0. Technical report, Université Paul Sabatier, Toulouse, 2005.

[49] A. S. Troelstra, D. van Dalen. *Constructivism in Mathematics: An Introduction*, v. 121 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, 1988.

[50] L. G. Valiant. General context-free recognition in less than cubic time. In *J. of Comput. and Syst. Sci.*, v. 10(2), pp. 308–314, 1975.

[51] P. Wadler. Propositions as types. *Commun. of the ACM*, v. 58(12), pp. 75–84, 2015.

[52] D. Younger. Recognition and parsing of context-free languages in time $O(n^3)$. In *Inf. and Comput.*, v. 10(2), pp. 189–208, 1967.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Tarmo Uustalu. His energy, active support, and scientific taste are an endless source of inspiration and good advice. I learned a lot from our collaboration and can only hope that it will continue.

I am grateful to my colleagues for creating an enjoyable atmosphere which made the Institute of Cybernetics a great place to work. Special thanks to the administrative staff who managed all the paperwork and allowed me to focus on research only.

I want to thank my wife and my sons for patience and keeping my motivation up.

Last but not least, I want to thank my mother and father for encouragements and care through all these years.

The work reported in this thesis was supported by the ERDF funded Estonian CoE project EXCS (3.2.0101.08-0013) and ICT national programme project "Coinduction" (3.2.1201.13-0029), the ESF funded Estonian Doctoral School in ICT (1.2.0401.09-0081), the Estonian Science Foundation grant no. 9475, and the Estonian Research Council personal research grant no. PUT763. I would also like to thank the Estonian IT Academy Programme for the stipends I received in 2012, 2013, and 2015.

Abstract

Context-free grammars are a widely used formalism in compiler construction for defining the syntactical structure of programming languages. In this thesis, our main goal is to implement and certify a parser generator for context-free languages. A parser generator takes a context-free grammar and returns a function that tries to find a parse tree for a given string. A certified parser generator delivers valid parse trees and will find one, if it exists, for the given context-free grammar.

There are different approaches to show that programs are correct: model checking, axiomatic semantics, expressive type systems. We are interested in constructive branches of logic. Proofs carried out within a constructive logic may be considered as programs in a functional language. This is important because of the possibility of extraction of the purported object from an existence proof. Our work is done in the Agda dependently typed programming language. Agda provides a single framework to write functional code and prove properties about it.

The main constituents of a context-free grammar are finite sets of terminals, nonterminals, and rules. Therefore, in the first part of the thesis, we investigate various encodings and properties of finiteness in constructive mathematics. A set is considered listable, if it can be completely enumerated in a list. We begin by showing that listable sets have decidable equality. This result allows us to conclude that certain basic variations of listability are logically equivalent to each other. We also develop a library of combinators to ease programming with finite sets. The library includes combinators for concise definition of functions on listable sets and a prover for quantified formulas over decidable properties on listable sets. Additionally, we propose that new listable sets can be defined concisely by listing a subset of a base set with decidable equality.

The second part of the thesis is devoted to parsing. We implement and certify the Cocke–Younger–Kasami algorithm, as it has a simple and elegant structure. Moreover, the algorithm allows one to parse general context-free languages. The relative simplicity contributes greatly to certifying the implementation. The naive recursive encoding leads to excessive recomputations, but we recover the efficient algorithm by introducing memoization. The refinement to the memoized version is done in a provably correctness-preserving manner.

The downside of the CYK parsing algorithm is that it requires context-free grammars in Chomsky normal form. The last part of the thesis focuses on normalization of context-free grammars. We divide normalization into four independent transformations. For each transformation, we prove that it achieves progress towards normality and also preserves the language of the grammar. We also show that the composition of transformations in the appropriate order converts any context-free grammar into its Chomsky normal form. Moreover, the proof of soundness of the normalization procedure is a function for converting any parse tree for the normalized grammar back into a parse tree for the original grammar.

Resümee

Kontekstivabad grammatikad on formalism, mida laialt kasutatakse programmeerimiskeelte süntaksi defineerimisel. Väitekirja peamiseks eesmärgiks on realiseerida ja sertifitseerida kontekstivabade grammatikate parsergeneraator. Parsergeneraator võtab kontekstivaba grammatika ja tagastab funktsiooni, mis stringide jaoks püüab leida parsimispuid. Sertifitseeritud parsergeneraator tagastab ainult õigesti moodustatud parsimispuid ja kindlasti leiab parsimispuu, kui see on olemas.

Leidub mitmeid viise, mis võimaldavad näidata programmide korrektsust: mudelikontroll, aksiomaatiline semantika, tugevad tüübisüsteemid. Me oleme huvitatud loogika konstruktiivsetest harudest. Konstruktiivse loogika tõestustest võib mõelda kui programmidest funktsionaalses programmeerimiskeeles. See on oluline, kuna eksistentsi tõestatusest on niisugusel puhul võimalik ekstraheerida konkreetne leiduv objekt. Meie töö on tehtud Agdas, mis on sõltuvate tüüpidega funktsionaalprogrammeerimise keel. Agda oluline tugevus seisneb selles, et ta pakub ühtset raamistikku nii programmeerimiseks kui ka omaduste tõestamiseks.

Kontekstivaba grammatika peamisteks komponentideks on lõplikud hulgad terminalidest, mitteterminalidest ja reeglitest. Seetõttu töö esimeses osas uurime me lõplike hulkade erinevaid definitsioone ja omadusi konstruktiivses matemaatikas. Hulk on lõplikult loetletav, kui eksisteerib lõplik loetelu, mis sisaldab kõik selle hulga elemendid. Me alustame tõestusega, et kõik loetletavad hulgad omavad lahenduvat võrdust. See tulemus võimaldab meid järeldada, et lõpliku loetletavuse teatud lihtsad variandid on kõik omavahel ekvivalentsed. Samuti arendame me kombinaatorite teegi, mis lihtsustab lõplike hulkadega programmeerimist. Teek sisaldab kombinaatoreid, mis võimaldavad lühidalt defineerida totaalseid funktsioone ja tõestada valemeid, mis on kvantifitseeritud üle lõplike hulkade. Lisaks näitame me, et uusi lõplikke hulki saab lühidalt defineerida lahenduva võrdusega baashulga alamhulga loetlemisega.

Teine osa väitekirjast on pühendatud parsimisele. Me otsustasime realiseerida ja sertifitseerida Cocke-Younger-Kasami algoritmi tema lihtsuse ja elegantsuse tõttu. CYK algoritm võimaldab parsida kõiki kontekstivabu keeli. Algoritmi suhteline lihtsus hõlbustab sertifitseerimist. Algoritmi naiivne rekursiivne versioon põhjustab liigseid korduvaid arvutusi, kuid soovitav efektiivne versioon on sellest hõlpsasti saavutatav memotabelite sissetoomisega. See peenendav üleminek on tõestatavalt korrektsust säilitav.

CYK algoritm töötab ainult grammatikatega, mis on Chomsky normaalkujul. Töö viimane osa on pühendatud grammatikate normaliseerimisele. Me jagame normaliseerimise neljaks sõltumatuks teisenduseks. Iga teisenduse kohta me tõestame, et ta saavutab teatud progressi normaalkuju suunas ning säilitab algse grammatika keele. Samuti tõestame me, et nende teisenduste kompositsioon sobivas järjekorras annab Chomsky normaalkuju. Konstruktiivsuse tõttu kujutab korrektsustõestus endast funktsiooni normaliseeritud grammatika mistahes parsimispuu konverteerimiseks algse grammatika parsimispuuks.

PUBLICATIONS

Paper I

D. Firsov, T. Uustalu. **Dependently typed programming with finite sets.** In *Proc. of 2015 ACM SIGPLAN Wksh. on Generic Programming, WGP '15 (Vancouver, BC, Aug. 2015)*, pp. 33–44. ACM Press, 2015.

Dependently Typed Programming with Finite Sets

Denis Firsov Tarmo Uustalu

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
{denis,tarmo}@cs.ioc.ee

Abstract

Definitions of many mathematical structures used in computer science are parametrized by finite sets. To work with such structures in proof assistants, we need to be able to explain what a finite set is. In constructive mathematics, a widely used definition is listability: a set is considered to be finite, if its elements can be listed completely. In this paper, we formalize different variations of this definition in the Agda programming language. We develop a toolbox for boilerplate-free programming with finite sets that arise as subsets of some base set with decidable equality. Among other things we implement combinators for defining functions from finite sets and a prover for quantified formulas over decidable properties on finite sets.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—correctness proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords certified programming; finite sets; dependently typed programming; Agda; Kuratowski finiteness; Bishop finiteness

1. Introduction

Many definitions of structures used in computer science are parametrized by finite sets. For example, in the theory of formal languages, a deterministic finite automaton is defined as a 5-tuple

$$\mathtt{M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)},$$

where $\mathtt{Q}$ is a finite set of states, Σ is a finite set of letters (alphabet), δ is a transition function from $\mathtt{Q} \times \Sigma$ to $\mathtt{Q}$, $\mathtt{q_0}$ is an initial state and $\mathtt{F}$ is a set of accepting states. To work with such concepts in proof assistants like Agda [13], which is the language we use in this paper, we need to be able to say what a finite set is.

One standard way to state that some set `X` is finite is to provide a list containing all elements of `X`. In our example, if the alphabet is binary (`Σ := 𝔹`,), then the list `false :: true :: []` together with a proof that every truth value is contained in this list establish finiteness of `Σ`. Another (equivalent) option is to provide a surjection from the set $[0..n)$ for some $n \in \mathbb{N}$. In our case, we can do with the function from $[0..2)$ that sends 0 to `false` and 1 to `true` together with a proof that this function is surjective.

In what follows, we define an example taken from quantum computing [12], the Pauli group on 1 qubit (with the global phase quotiented out), as a datatype with 4 nullary constructors. We also implement equality decision and the group operation to highlight the weak points of this straightforward approach and show the boilerplate code that we would like to reduce.

1.1 Extended Example

A finite set like the Pauli group can be defined as a datatype with a nullary constructor for each element:

```
data Pauli : Set where
  X : Pauli
  Y : Pauli
  Z : Pauli
  I : Pauli
```

The constructors `X`, `Y`, `Z`, and `I` denote the four distinct elements of the set `Pauli`.

To show that `Pauli` is finite (so that one can, for example, iterate through all elements), we can provide a list:

```
listPauli : List Pauli
listPauli = X :: Y :: Z :: I :: []
```

We can prove that the list is complete:

```
allPauli : (x : Pauli) → x ∈ listPauli
allPauli X = here
allPauli Y = there here
allPauli Z = there (there here)
allPauli I = there (there (there here))
```

(Here, `here` is a proof of `x ∈ x :: xs`; and `there p` is a proof of `x ∈ y :: xs`, if `p` is a proof that `x ∈ xs`.)

We could also prove that this list does not contain duplicates, but this is not mandatory.

We continue our example by implementing equality decision for elements of `Pauli`:

```
_≡P?_ : (x₁ x₂ : Pauli) → x₁ ≡ x₂ ⊎ ¬ (x₁ ≡ x₂)
X ≡P? X = inj₁ refl
X ≡P? Y = inj₂ λ()
X ≡P? Z = inj₂ λ()
X ≡P? I = inj₂ λ()
Y ≡P? X = inj₂ λ()
Y ≡P? Y = inj₁ refl
Y ≡P? Z = inj₂ λ()
Y ≡P? I = inj₂ λ()
Z ≡P? X = inj₂ λ()
Z ≡P? Y = inj₂ λ()
Z ≡P? Z = inj₁ refl
Z ≡P? I = inj₂ λ()
```


WGP'15, August 30, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3810-3/15/08...$15.00
http://dx.doi.org/10.1145/2808098.2808102

```
I ≡P? X = inj₂ λ()
I ≡P? Y = inj₂ λ()
I ≡P? Z = inj₂ λ()
I ≡P? I = inj₁ refl
```

(¬ P = P → ⊥, where ⊥ is the empty set; X ⊎ Y denotes the disjoint sum of X and Y. () is called the absurd pattern and denotes impossibility of a pattern.)

To conclude our example, we define the group operation:

```
_·_ : Pauli → Pauli → Pauli
X · X = I
X · Y = Z
X · Z = Y
Y · X = Z
Y · Y = I
Y · Z = X
Z · X = Y
Z · Y = X
Z · Z = I
x · I = x
I · x = x
```

And we prove that it is commutative:

```
·-comm : (x₁ x₂ : Pauli) → x₁ · x₂ ≡ x₂ · x₁
·-comm X X = refl
·-comm X Y = refl
·-comm X Z = refl
·-comm X I = refl
·-comm Y X = refl
·-comm Y Y = refl
·-comm Y Z = refl
·-comm Y I = refl
·-comm Z X = refl
·-comm Z Y = refl
·-comm Z Z = refl
·-comm Z I = refl
·-comm I X = refl
·-comm I Y = refl
·-comm I Z = refl
·-comm I I = refl
```

It is important to realize that `refl` takes different implicit arguments in different lines of the code above, so it cannot be shortened to just one line `·-comm _ _ = refl`. Actually, the code shown is the shortest "direct" proof and requires full pattern matching. It is easy to see that an associativity proof requires 64 lines of code.

We can see that the straightforward way of defining a finite set as an enumeration type has a number of shortcomings:

1. When defining `Pauli` and `listPauli`, we effectively listed all elements twice.
2. The proof of `allPauli` is verbose and dependent on the order of elements in the list `listPauli`. All three definitions (`Pauli`, `listPauli`, `allPauli`) must be kept consistent at all times, when modifying the code.
3. The equality decider is not derived automatically and the manual definition is verbose. The same would apply to duplicate-freeness decision, if we wanted to implement it.
4. The proof of commutativity of the `_·_` operation is dull, but cannot be compressed.

Alternatively, to show that `Pauli` is finite, we can provide a surjection from an initial segment of natural numbers. Let us first introduce a family of sets for initial segments of the set of all natural numbers. `Fin n` represents the set of first `n` natural numbers, i.e., the set of all numbers smaller than `n`.

```
data Fin : ℕ → Set where
  fzero : {n : ℕ} → Fin (suc n)
  fsuc  : {n : ℕ} → Fin n → Fin (suc n)
```

(In Agda, an argument enclosed in curly braces is implicit. The Agda type-checker will try to figure it. If an argument cannot be inferred, it must be provided explicitly.) `fzero` is smaller than `suc n` for any `n` and, if `i` is smaller than `n`, then `fsuc i` is smaller than `suc n`. As there is no number smaller than zero, `Fin zero` is empty. (When there are no possible constructor patterns for a given argument, one can pattern match on it with the absurd pattern `()`.)

Now, to show that `Pauli` is finite, we can define a function from `Fin 4` to `Pauli`:

```
f2p : Fin 4 → Pauli
f2p fzero = X
f2p (fsuc fzero) = Y
f2p (fsuc (fsuc fzero)) = Z
f2p (fsuc (fsuc (fsuc fzero))) = I
f2p (fsuc (fsuc (fsuc (fsuc ()))))
```

We can also define a function in the converse direction:

```
p2f : Pauli → Fin 4
p2f X = fzero
p2f Y = fsuc fzero
p2f Z = fsuc (fsuc fzero)
p2f I = fsuc (fsuc (fsuc fzero))
```

This allows us show that `f2p` is surjective by establishing

```
f2p-surj : (x : Pauli) → f2p (p2f x) ≡ x
f2p-surj X = refl
f2p-surj Y = refl
f2p-surj Z = refl
f2p-surj I = refl
```

In fact, `f2p` is not only a surjection, but even a bijection, but this is not mandatory for finiteness. We have once more established that `Pauli` is finite, however we introduced even more dependencies than when defining `Pauli`, `listPauli`, and `allPauli`. Namely, now the definitions of `Pauli`, `f2p`, `p2f`, and `f2p-surj` must all be kept in agreement with each other.

In this paper, we set out to explore finite sets in the constructive and dependently typed setting of Agda and develop an infrastructure for clean programming with finite sets.

In Section 2, we give some basic definitions regarding decidable propositions and also introduce effective squashing for such propositions.

In Section 3, we introduce some notions of constructive finiteness of a set and of a subset of a set and explore how they are related to each other. We present some conditions under which equality on a finite set is decidable. Furthermore, we show that there are finite subsets for which equality is undecidable.

Section 4 is devoted to a pragmatic approach for programming with finite sets that arise as subsets of a base set with decidable equality. In this approach, we are able to define a finite subset by listing its elements once and automatically derive completeness and decidable equality.

In Section 5, we show that the union, intersection, product and disjoint sum of finite sets are finite.

In Section 6.1, we show that functions from finite sets can be defined via tables. After that in Section 6.2, we introduce the notion of predicate matching and show how it can be used for defining functions on finite sets.

In Section 7, we implement a prover for quantified formulas over decidable properties on finite sets.

We used Agda 2.4.2.2 and Agda Standard Library 0.9 for this development. The full Agda code of this paper can be found at `http://cs.ioc.ee/~denis/finset/`.

2. Basic Definitions

The predicate `All X` states that a given list `xs` contains all elements of a set `X` (duplicates being allowed):

```
All : (X : Set) → List X → Set
All X xs = (x : X) → x ∈ xs
```

A proposition `P` is called decidable, if there is a proof of either `P` or not `P`:

```
data Dec (P : Set) : Set where
  yes : P   → Dec P
  no  : ¬ P → Dec P
```

(Here `yes` and `no` are two constructors of the datatype `Dec P`. The former takes a proof of `P` as its argument, while the latter takes a proof of $\neg$ `P`.)

Now, we say that a set `X` has decidable equality, if there is a function sending any elements x_1 and x_2 of `X` to a proof of `Dec` $(x_1 \equiv x_2)$:

```
DecEq : (X : Set) → Set
DecEq X = (x₁ x₂ : X) → Dec (x₁ ≡ x₂)
```

With these notations, the type of `_≡P?_` from Section 1 can be abbreviated to `DecEq Pauli`.

Similarly, we can define decidable list membership:

```
DecIn : (X : Set) → Set
DecIn X = (x : X) → (xs : List X) → Dec (x ∈ xs)
```

A proof of `DecIn X` is a function that, for any element `x : X` and a list `xs : List X`, returns a proof of either `x ∈ xs` or its negation. It is easy to verify that `DecEq X` and `DecIn X` are equivalent, namely:

```
deq2din : {X : Set} → DecEq X → DecIn X

din2deq : {X : Set} → DecIn X → DecEq X
```

We also define a notion of a proposition `P` being a mere proposition:

```
Prop : Set → Set
Prop P = (p₁ p₂ : P) → p₁ ≡ p₂
```

It says that `P` can have at most one proof.

Another basic predicate is `NoDup`, which expresses that a given list `xs` is duplicate-free:

```
NoDup : {X : Set} → List X → Set
NoDup {X} xs = (x : X) → Prop (x ∈ xs)
```

Duplication-freeness of `xs` is the same as there being at most one proof of membership in `xs` for every `x : X`.

If `X` has decidable equality, then `All X` and `NoDup` are decidable:

```
deq2dall : {X : Set} → DecEq X
  → (xs : List X) → Dec (All X xs)

deq2dnd : {X : Set} → DecEq X
  → (xs : List X) → Dec (NoDup xs)
```

If `P` is decidable, we can effectively define a squashed version of `P` (i.e., quotient `P` by the total equivalence relation):

```
‖_‖ : {P : Set} → Dec P → Set
‖ yes _ ‖ = ⊤
‖ no  _ ‖ = ⊥
```

(Here $\top$ is the unit type: a singleton type with a unique element `tt`.) Note that we are squashing `P`, not `Dec P`, however, we make use of a proof that `P` is decidable. For example, we can observe that the type

```
X ∈ X :: Y :: X :: []
```

is decidable. Moreover, there are two different proofs:

```
prf₁ : Dec (X ∈ X :: Y :: X :: [])
prf₁ = yes here

prf₂ : Dec (X ∈ X :: Y :: X :: [])
prf₂ = yes (there (there here))
```

So we can squash the type `X ∈ X :: Y :: X :: []` in two different ways: ‖ `prf₁` ‖ or ‖ `prf₂` ‖, but both evaluate to $\top$.

It is easy to see that any two elements of a squashed type are equal:

```
propSq : {P : Set} → (d : Dec P) → Prop ‖ d ‖
```

It is also important to note that one can always get a proof of `P`, if the squashed version is inhabited:

```
fromSq : {P : Set} → (d : Dec P) → {‖ d ‖} → P
```

We have made the third argument (of type ‖ `d` ‖) implicit, since if `d` proves `Dec P`, then the only possible value is the unique element `tt : ⊤` and the type-checker can derive it automatically.

3. Finiteness Constructively

3.1 Listable Sets

The best known and most used constructive notion of finiteness of a set is listability (also sometimes called Kuratowski finiteness, although Kuratowski [11] phrased his definition in different terms): a set is finite, if its elements can be completely listed:

```
Listable : (X : Set) → Set
Listable X = Σ[ xs ∈ List X ]
               All X xs
```

(In Agda, `Σ[ a ∈ A ] B a` is the type of dependent pairs of an element `a` of type `A` and an element of type `B a`. Note the unfortunate and confusing use of $\in$ instead of `:` for typing the bound variable in this notation.)

A close alternative idea is to require a surjection from an initial segment of the set of natural numbers:

```
FinSurj : (X : Set) → Set
FinSurj X = Σ[ n ∈ ℕ ]
              Σ[ fromFin ∈ (Fin n → X) ]
              Σ[ toFin   ∈ (X → Fin n) ]
              ((x : X) → fromFin (toFin x) ≡ x)
```

The two notions are equivalent:

```
surj2lstbl : {X : Set}
  → FinSurj X → Listable X

lstbl2surj : {X : Set}
  → Listable X → FinSurj X
```

It is clear that, from listability of a set, one can learn an upper bound on the number of its elements. (But in fact one can learn also the actual cardinality, just wait a little.)

An a priori trimmer version of listability (sometimes called Bishop-finiteness [6]) forbids duplicates:

```
ListableNoDup : (X : Set) → Set
ListableNoDup X = Σ[ xs ∈ List X ]
                    All X xs ×
                    NoDup xs
```

Alternatively, one may require a bijection from an initial segment of the set of natural numbers:

```
FinBij : (X : Set) → Set
FinBij X = Σ[ n ∈ ℕ ]
           Σ[ fromFin ∈ (Fin n → X) ]
           Σ[ toFin ∈ (X → Fin n) ]
           ((x : X) → fromFin (toFin x) ≡ x) ×
           ((i : Fin n) → toFin (fromFin i) ≡ i)
```

These two notions of finiteness are also equivalent:

```
bij2lstblnd : {X : Set}
  → FinBij X → ListableNoDup X

lstblnd2bij : {X : Set}
  → ListableNoDup X → FinBij X
```

Quite clearly, from duplicate-free listability of a set, one can extract its exact cardinality.

It is less obvious that all four notions of finiteness are equivalent. The reason is that equality on a listable set is decidable:

```
lstbl2deq : {X : Set} → Listable X → DecEq X
```

We give the main idea behind the implementation of `lstbl2deq`. If a set `X` is listable, then there exist a list `xs` and a function `cmplt` that, for any element `x : X`, returns a proof that `x ∈ xs`. You can think of this proof as a position of `x` in `xs`. Therefore, for any value `x`, there is a split of `xs` such that $\texttt{xs} \equiv \texttt{xs}_1 \texttt{ ++ x :: xs}_2$. Now, if we need to check whether $\texttt{x}_1 \texttt{ : X}$ and $\texttt{x}_2 \texttt{ : X}$ are equal, we can proceed as follows. We ask `cmplt` for two splits of `xs`: `cmplt` $\texttt{x}_1$ gives a split $\texttt{xs} \equiv \texttt{xs}_1 \texttt{ ++ x}_1 \texttt{ :: xs}_2$ and `cmplt` $\texttt{x}_2$ gives a split $\texttt{xs} \equiv \texttt{xs}' \texttt{ ++ x}_2 \texttt{ :: xs}''$. Next, it is clear that if

$$\texttt{length xs}_1 \equiv \texttt{length xs}'$$

then $\texttt{x}_1 \equiv \texttt{x}_2$. But what if $\texttt{length xs}_1 \not\equiv \texttt{length xs}'$? If the list `xs` were guaranteed to be duplicate-free, then this would immediately imply that $\texttt{x}_1 \not\equiv \texttt{x}_2$, since there would then be a bijection between positions of `xs` and elements of `X`. However, in the presence of duplicates, this argument does not work. Instead, we observe that `cmplt` is a function of type $\texttt{(x : X)} \to \texttt{x} \in \texttt{xs}$. Therefore, for equal elements of `X`, it must deliver equal results. With this observation, we can argue that if $\texttt{length xs}_1$ is not equal to $\texttt{length xs}'$ then $\texttt{x}_1 \not\equiv \texttt{x}_2$. Indeed, if $\texttt{x}_1 \equiv \texttt{x}_2$, then $\texttt{cmplt x}_1$, and $\texttt{cmplt x}_2$ must give the same split contradicting $\texttt{length xs}_1 \not\equiv \texttt{length xs}'$.

As soon as list membership is decidable, we can implement removal of duplicates:

```
remDup : {X : Set} → DecIn X → List X → List X
```

We show that `remDup` is complete. Namely, if some element belongs to the list, then at least one copy of that element is preserved by `remDup`:

```
remDupComplete : {X : Set} → (_∈?_ : DecIn X)
  → (x : X) → (xs : List X)
  → x ∈ xs → x ∈ remDup ∈? xs
```

`remDup` is also sound—the resulting list does not contain any new elements:

```
remDupSound : {X : Set} → (_∈?_ : DecIn X)
  → (x : X) → (xs : List X)
  → x ∈ remDup ∈? xs → x ∈ xs
```

And most importantly, the resulting list is free of duplicates:

```
remDupProgress : {X : Set} → (_∈?_ : DecIn X)
  → (xs : List X) → NoDup (remDup ∈? xs)
```

Now, with `lstbl2deq` and `deq2din`, we can prove that `Listable X` implies `ListableNoDup X`, and the converse is a triviality:

```
lstbl2lstblnd : {X : Set}
  → Listable X → ListableNoDup X

lstblnd2lstbl : {X : Set}
  → ListableNoDup X → Listable X
```

It is worth noticing that the proof `lstbl2deq` also provides an alternative definition of an equality decider for listable types like `Pauli` from Section 1:

```
listablePauli : Listable Pauli
listablePauli = listPauli , allPauli

deqPauli : DecEq Pauli
deqPauli = lstbl2deq listablePauli
```

Remember that the direct approach for defining decidable equality on `Pauli` required us 4^2 lines of code.

3.2 Listable Subsets

A special case of sets are those defined as a subset of a larger set. Here we have more variations of finiteness.[1].

A subset of a base set `U` carved out by a predicate `P : U → Set` is called subfinite, if there is a list containing all elements of `U` that satisfy `P` (we call this property completeness):

```
ListableJunkSub : (U : Set) → (U → Set) → Set
ListableJunkSub U P = Σ[ xs ∈ List U ]
                      ((x : U) → P x → x ∈ xs)
```

This notion of finiteness (which can only be formulated for subsets of some base set, not for general sets) allows `xs` to contain also elements not satisfying `P`. Therefore, we cannot even know whether the subset is empty. But we have an immediate upper bound on the number of elements in the subset: it is the length of the list `xs`.

A stronger notion of finiteness requires also soundness, i.e., a proof that an element of `U` belongs to `xs` only if it satisfies the predicate `P` (duplicates are still allowed):

```
ListableSub : (U : Set) → (U → Set) → Set
ListableSub U P = Σ[ xs ∈ List U ]
                    ((x : U) → P x → x ∈ xs) ×
                    ((x : U) → x ∈ xs → P x)
```

A listable subset can be checked for emptiness:

```
empty? : {U : Set}{P : U → Set}
  → (p : ListableSub U P)
  → Dec ((x : U) → ¬ x ∈? proj₁ p))
```

Listable sets are a special case of listable subsets:

```
lstbl2lsub : {U : Set}
  → Listable U → ListableSub U (λ _ → ⊤)

lsub2lstbl : {U : Set}
  → ListableSub U (λ _ → ⊤) → Listable U
```

The always true predicate $\texttt{(λ \_ → ⊤)}$ gives us the whole set `U` as the subset, i.e., the base set `U` must itself be listable. This is a special case of the situation where `P` has at most one proof for every element `x` of `U` (`P x` is a mere proposition):

```
prop2lstbl2lsub : {U : Set}{P : U → Set}
```

[1] We will generally speak of finiteness of a subset without actually constructing this subset as a set in its own right, since that would require us to be able to squash arbitary propositions, not just decidable ones.

```
  → ((x : U) → Prop (P x))
  → Listable (Σ[ x ∈ U ] P x)
  → ListableSub U P

prop2lsub2lstbl : {U : Set}{P : U → Set}
  → ((x : U) → Prop (P x))
  → ListableSub U P
  → Listable (Σ[ x ∈ U ] P x)
```

3.3 Decidability of Equality on Listable Subsets

Let us define decidability of equality on the subset of `U` determined by `P` as decidability of equality on `U` restricted to the elements satisfying `P`:

```
DecEqSub : (U : Set) → (P : U → Set) → Set
DecEqSub U P
  = (x₁ x₂ : U) → P x₁ → P x₂ → Dec (x₁ ≡ x₂)
```

In Section 3, we showed that listability of `X` implies decidable equality on `X`. Now we give a more general version of that property, namely, if, for any `x : U` there is at most one proof of `P x`, then equality on the subset given by `U` and `P` is decidable.

```
deqLstblSub1 : {U : Set}
   → (P : U → Set)
   → ListableSub U P
   → ((x : U) → Prop (P x))
   → DecEqSub U P
```

The strategy of implementing `deqLstblSub1` is similar to the strategy of implementing `lstbl2deq`. If `P` defines a listable subset of `U`, then we have a list `xs` containing all elements of `U` such that `P`. We also have a proof of completeness of `xs`:

```
cmplt : (x : U) → P x → x ∈ xs.
```

If we want to check two elements x_1 and x_2 for equality, then we are also given proofs p_1 : P x_1 and p_2 : P x_2. Clearly, if `cmplt` x_1 p_1 and `cmplt` x_2 p_2 induce the same splits of `xs`, namely, `xs` $\equiv$ `xs`$_1$ `++` x_1 `::` `xs`$_2$, `xs` $\equiv$ `xs`$'$ `++` x_2 `::` `xs`$''$ and `length xs`$_1$ $\equiv$ `length xs`$'$, then $x_1 \equiv x_2$. However, in the case when the splits are different, we cannot use the argument that, since `cmplt` is a function, there is only one split for each element. The reason is that, generally, `cmplt x` may deliver different splits for different proofs of `P x`. However, we have required that there is a unique proof of `P x` for any `x`. Finally, we can conclude that, if `length xs`$_1$ $\not\equiv$ `length xs`$_2$, then $x_1 \not\equiv x_2$.

Actually, in this situation of `P` being a mere proposition, the intended subset can be explicitly defined as the set `Σ[ x ∈ U ] P x`, and we have decidable equality on this set:

```
deqLstblSub1' : {U : Set}
   → (P : U → Set)
   → ListableSub U P
   → ((x : U) → Prop (P x))
   → (xp₁ xp₂ : Σ[ x ∈ U ] P x)
   → Dec (xp₁ ≡ xp₂)
```

Equality on the subset is also decidable, if `P` is decidable:

```
deqLstblSub2 : {U : Set}
   → (P : U → Set)
   → ((x : U) → Dec (P x))
   → ListableSub U P
   → DecEqSub U P
```

A further variation says that, if we know that the list of all elements of `U` satisfying the predicate `P` is duplicate-free, then we also have decidable equality on the subset:

```
deqLstblSub3 : {U : Set}
   → (P : U → Set)
   → (p : ListableSub U P)
   → NoDup (proj₁ p)
   → DecEqSub U P
```

We conclude with a proof that there is no function turning any proof of `ListableSub U P` into a decider of equality on elements of `U` satisfying `P`:

```
deqLstblSub4 : {U : Set}
   → (P : U → Set)
   → ListableSub U P
   → DecEqSub U P
deqLstblSub4 = ???
```

Let us define the following list of functions from booleans to booleans:

```
listB2B : List (Bool → Bool)
listB2B = fun₁ :: fun₂ :: fun₃ :: []
  where
    fun₁ : Bool → Bool
    fun₁ _ = true

    fun₂ : Bool → Bool
    fun₂ _ = false

    fun₃ : Bool → Bool
    fun₃ b = if b then true else true
```

The list `listB2B` consists of three functions. The functions `fun`$_1$ and `fun`$_3$ always return `true`, however they are not propositionally equal, unless we assume function extensionality. The function `fun`$_2$ always returns `false`.

Then we specify a subset of functions of type `Bool → Bool` by the following predicate `B2B`:

```
B2B : (Bool → Bool) → Set
B2B f = f ∈ listB2B
```

Next, we prove that the predicate `B2B` defines a listable subset. Clearly, it is just the set of functions from the list `listB2B`:

```
listableB2B : ListableSub (Bool → Bool) B2B
listableB2B = listB2B , (λ x p → p) , (λ x p → p)
```

So, now we could try to write a function that decides equality of elements of `listableB2B`:

```
deqB2B : (f₁ f₂ : Bool → Bool)
   → B2B f₁
   → B2B f₂
   → Dec (f₁ ≡ f₂)
deqB2B = ???
```

Given `f`$_1$ and `f`$_2$ together with `p`$_1$: `B2B f`$_1$ and `p`$_2$: `B2B f`$_2$, we can pattern-match on `p`$_1$ and `p`$_2$. Some cases are unproblematic: e.g., if `p`$_1$ `= here` and `p`$_2$ `= here`, then `f`$_1$ `= fun`$_1$ and `f`$_2$ `= fun`$_1$, so it is trivial that `f`$_1$ $\equiv$ `f`$_2$. Similarly, if `p`$_1$ `= here` and `p`$_2$ `= there here`, then `f`$_1$ `= fun`$_1$ and `f`$_2$ `= fun`$_2$ and hence $\neg$ `(f`$_1$ $\equiv$ `f`$_2$`)`. But there is the critical case of `p`$_1$ `= here` and `p`$_2$ `= there (there here)`. Then `f`$_1$ `= fun`$_1$ and `f`$_2$ `= fun`$_3$ and there is simply no correct answer to return, as the two functions are not equal propositionally (unless function extensionality is assumed), but also not inequal.

We have argued that it is impossible to prove `deqLstblSub4`. This also implies that the notion of a listable set is stronger than the notion of a listable subset, which in turn is stronger than the notion of a subset listable with junk.

4. Pragmatic Finite Sets

In this section, we aim at a pragmatic approach to programming with finite sets. Our objective is to be able to specify a finite set by listing the intended elements just once. From specification, we want to obtain a listable set with no additional work. Our solution is to specify the finite set as a subset of some base set with decidable equality.

4.1 Motivation and Definition

In Section 1, we saw that the straightforward approach to defining the Pauli group as a datatype with nullary constructors and proving that it is finite required us to list the elements of `Pauli` multiple times and also provide verbose proofs of completeness and decidability of equality. Next, we go through a number of steps, to motivate a more pragmatic approach.

As we have seen, a predicate `P` on a base set `U` defines a subset. If there is a list of elements containing all elements of `U` satisfying `P` and no others, then we have a listable subset:

```
step1 : {U : Set} → (P : U → Set)
  → (xs : List U)
  → ((x : U) → x ∈ xs → P x)
  → ((x : U) → P x → x ∈ xs)
  → ListableSub U P
```

Next, we observe that we can create a listable subset from any list `xs` over `U` by taking `P` to be `(λ x → x ∈ xs)`:

```
step2 : {U : Set} → (xs : List U)
  → ListableSub U (λ x → x ∈ xs)
step2 = xs , (λ x i → i) , (λ x i → i)
```

A good thing is that the proofs of soundness and completeness are now trivial. But the elements of the subset are dependent pairs of an element `x` of `U` and a proof of membership (position) of `x` in `xs`.

Next we can ask for decidable list membership on `U` to be able to effectively squash sets `x ∈ xs`:

```
step3 : {U : Set} → (_∈?_ : DecIn U)
  → (xs : List U)
  → ListableSub U (λ x → ‖ x ∈? xs ‖)
```

Now an element of the subset is a pair of an element of `U` and an element of a squashed type (which, if it exists, is unique!).

By theorem `prop2lsub2lstbl` from Section 3.2, the type

```
Σ[ x ∈ U ] ‖ x ∈? xs ‖
```

must be listable.

Given these considerations, we can define a datatype of descriptions of finite sets as subsets of a base set with decidable equality:

```
data FinSubDesc (U : Set) (eq : DecEq U) :
                              Bool → Set where

  fsd-plain : List U → FinSubDesc U eq true
  fsd-nodup : (xs : List U) → {‖ nd? xs ‖}
    → FinSubDesc U eq false
      where
        nd? = deq2dnd eq
```

The datatype introduced is parametrized by a base set `U`, a decider `eq` of equality on `U`, and is also indexed by a boolean flag `b` that indicates whether the underlying list of elements is allowed to contain duplicates. There are two constructors. The constructor `fsd-plain` takes a list `xs` of elements of `U` as an argument. The constructor `fsd-nodup` accepts a list `xs` as an argument only if it is duplicate-free. It has also another, implicit argument, of a squashed type. This type is inhabited if and only if `xs` contains no duplicates. In other words, if `xs` is duplicate-free, then the type of the implicit argument evaluates to the unit type and its value can be inferred automatically. If `xs` contains duplicates, then the type of the implicit argument evaluates to $\bot$ and no value can be provided for it.

There are pragmatic reasons to have two constructors for `FinSubDesc`. If the user creates a relatively small subset of elements ($\leq$ 10000) using `fsd-nodup`, then the type-checker can feasibly check that there are no duplicates. However, if the number of elements is larger, then the price for maintaining the invariant of no duplicates becomes too high. Remember that the complexity of checking duplicate-freeness is quadratic in the length of the list.

We can now define the Pauli group as a subset of the set of all characters:

```
MyPauli : FinSubDesc Char _≡C?_ false
MyPauli
  = fsd-nodup ('X' :: 'Y' :: 'Z' :: 'I' :: [])
```

Since the list provided is without duplicates, the type of the implicit argument

```
‖ nd? ('X' :: 'Y' :: 'Z' :: 'I' :: []) ‖
```

is evaluated to $\top$ by the type-checker and the value for this argument is derived automatically.

On the other hand, the following definition is rejected by the type-checker, since `'X'` is listed twice and the type of the implicit argument is evaluates to $\bot$:

```
MyPauliBad : FinSubDesc Char _≟?_ false
MyPauliBad
  = fsd-nodup ('X' :: 'Y' :: 'Z' :: 'I' :: 'X' :: [])
                                              {???}
```

The hole needs to be filled with a proof of $\bot$, which is impossible. However, we can drop the requirement of no duplicates (note the change in the type):

```
MyPauliFixed : FinSubDesc Char _≟?_ true
MyPauliFixed
  = fsd-plain ('X' :: 'Y' :: 'Z' :: 'I' :: 'X' :: [])
```

Now, we can define the actual set that a finite subset description denotes:

```
toList : {U : Set}{eq : DecEq U}{b : Bool}
  → FinSubDesc U eq b → List U
toList (fsd-plain xs) = xs
toList (fsd-nodup xs) = xs

Elem : {U : Set}{eq : DecEq U}{b : Bool}
  → FinSubDesc U eq b → Set
Elem {U} {eq} D
  = Σ[ x ∈ U ] ‖ x ∈? toList D ‖
    where
      _∈?_ = deq2din eq
```

So an element of type `Elem D` for some finite subset description `D` is a dependent pair of an element `x` of `U` together with a squashed proof that `x` belongs to the list of elements defining the subset. Using the squashed membership type allows us to ignore the exact position(s) of the element in the list.

For example, we could refer to one of the elements of `MyPauli` as the `identity`:

```
identity : Elem MyPauli
identity = ('I' , _)
```

The second component of the pair (the type-checker infers that it must be `tt`) is actually a squashed proof of the fact that `I` belongs to the set `MyPauli`. Without squashing, we would need to refer to

I by its position, namely,

```
('I', there (there (there here))).
```

Clearly, we want to avoid such fragile dependencies.

On the other hand, the type-checker will accept a non-element of the list only if the user manages to provide a proof of $\bot$.

```
bad : Elem MyPauli
bad = ('W' , ???)
```

4.2 Finite Subsets are Listable

Our next step is to show that, for all `D : FinSubDesc U eq b`, the corresponding subset of `U`, namely, `Elem D`, is listable.

First, we generate a list of elements of `Elem D`:

```
listElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (D : FinSubDesc U eq b)
  → List (Elem D)
```

Second, we show that `listElem D` is complete, it contains all elements of `Elem D`:

```
allElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (D : FinSubDesc U eq b)
  → (xp : Elem D) → xp ∈ listElem D
```

Third, we observe that `listElem D` does not introduce any duplicates:

```
ndElem : {U : Set}{eq : DecEq U}
  → (D : FinSubDesc U eq false)
  → NoDup (listElem D)
```

Finally, we show that `Elem D` is listable:

```
lstblElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (D : FinSubDesc U eq b)
  → Listable (Elem D)
lstblElem D = listElem D , allElem D
```

This also implies decidable equality on `Elem D`:

```
deqElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (f : FinSubDesc U eq b)
  → DecEq (Elem D)
deqElem D = lstbl2deq (lstblElem D)
```

4.3 Finite Subsets from Lists

Now, we implement a function `fromList` which is parametrized by the boolean `b`, so that user could decide if the duplicates should be removed from the resulting finite subset:

```
fromList : {U : Set} → (eq : DecEq U)
  → (b : Bool) → List U → FinSubDesc U eq b
```

Basic set operations can now be defined on the underlying lists of finite subsets. For example, the union is defined by concatenating the underlying lists of argument subsets:

```
_∪_ : {U : Set}{eq : DecEq U}
  → {b₁ b₂ : Bool}
  → FinSubDesc U eq b₁
  → FinSubDesc U eq b₂
  → FinSubDesc U eq (b₁ ∧ b₂)
_∪_ {eq = eq} D₁ D₂
  = fromList eq _ (toList D₁ ++ toList D₂)
```

Here is an example:

```
MyNats1 = fsd-nodup (1 :: 3 :: [])
MyNats2 = fsd-nodup (1 :: 6 :: [])
p : MyNats1 ∪ MyNats2 ≡ fsd-nodup (1 :: 3 :: 6 :: [])
p = refl
```

4.4 Finite Subset Monad

Finite subsets (of sets with decidable equality) are monad. We explicate this structure on the level of `FinSubDesc`:

```
return : {U : Set}{eq : DecEq U}{b : Bool}
   → U → FinSubDesc U eq b

bind : {U V : Set}{eqU : DecEq U}
  → {eqV : DecEq V}{bU bV : Bool}
  → FinSubDesc U eqU bU
  → (U → FinSubDesc V eqV bV)
  → (b : Bool)
  → FinSubDesc V eqV b
```

A peculiarity of `return` and `bind` here is that they can work in two different modes. If the boolean argument provided is `false`, then duplicates will be removed the resulting finite subset description, otherwise not. This allows the user to tune the monadic code for the efficiency.

Wadler [16] identifies the structure needed for comprehending monads. The missing bit is mzero:

```
mzero : {U : Set}{eq : DecEq U}{b : Bool}
  → FinSubDesc U eq b
mzero {b = true}  = fsd-plain []
mzero {b = false} = fsd-nodup []
```

Using `mzero`, we define a conditional `if_then_` and also some syntactic sugar for `bind`:

```
if_then_ : {U : Set}{eq : DecEq U}{b : Bool}
  → Bool → FinSubDesc U eq b
  → FinSubDesc U eq b
if b then c = if b then c else mzero

syntax bind A (λ x → B) b
  = for x ∈ A as b do B
```

As a result, we can write set comprehension code in for-loop style. Let us look at an example. Mathematically, the intersection of sets `X` and `Y` is defined as:

$$X \cap Y = \{x \mid x \in X, y \in Y, x = y\}$$

With the combinators and syntactic sugar defined above, we can write the following definition of subset intersection with comprehensions:

```
_∩_ : {U : Set}{eq : DecEq U} {b₁ b₂ : Bool}
  → FinSubDesc U eq b₁ → FinSubDesc U eq b₂
  → FinSubDesc U eq (b₁ ∧ b₂)
_∩_ {eq = _≡?_} X Y =
         for x ∈ X as _  do
           for y ∈ Y as true do
             if ⌊ x ≡? y ⌋ then return x
```

5. Combinators

In this section, we define some general combinators for listable subsets. The simplest combinator is for taking the union of two listable subsets of the same base set:

```
union : {U : Set}{P Q : U → Set}
  → ListableSub U P
  → ListableSub U Q
  → ListableSub U (λ x → P x ⊎ Q x)
```

The definition just concatenates the underlying lists of the two subsets and then adapts the proofs of completeness and soundness.

The intersection of two listable subsets is trickier, since it cannot be defined generally for two arbitrary subsets. The reason is simple, we need somehow to find the common elements. One possibility is to ask equality on `U` to be decidable:

```
intersection' : {U : Set}{P Q : U → Set}
  → DecEq U
  → ListableSub U P
  → ListableSub U Q
  → ListableSub U (λ x → P x × Q x)
```

But this assumption can be weakened by only asking one of the predicates to be decidable.

```
intersection : {U : Set}{P Q : U → Set}
  → ((x : U) → Dec (P x))
  → ListableSub U P
  → ListableSub U Q
  → ListableSub U (λ x → P x × Q x)
```

This a weaker condition, because, if `U` has decidable equality, then `ListableSub U P` implies decidability of `P`:

```
deq2lstbl2dp : {U : Set}{P : U → Set}
  → DecEq U
  → ListableSub U P
  → (x : U) → Dec (P x)
```

We also prove that the product and the disjoint sum of listable subsets of two base sets are listable subsets of the product/disjoint sum of the base sets:

```
product : {U : Set}{P : U → Set}
  → {V : Set}{Q : V → Set}
  → ListableSub U P
  → ListableSub V Q
  → ListableSub (U × V) < P , Q >

sum : {U : Set}{P : U → Set}
  → {V : Set}{Q : V → Set}
  → ListableSub U P
  → ListableSub V Q
  → ListableSub (U ⊎ V) [ P , Q ]
```

6. Function Definition

This section describes two different approaches for defining functions from finite sets.

We observe that, if we want to define arbitrary functions from some finite `X` to `Y`, then we must be able to compare the elements of `X` and also for the function to be total we need the complete list of those. Therefore, the right notion of finiteness for `X` is listability (`Listable X`).

6.1 Tabulation

To define a function of type `f : X → Y` for some listable `X`, we could explicitly provide a list of pairs `(x , y)`. For example, if `X = { ▲ , ♦ , ■ }` and `Y = ℕ` then the list

```
(▲ , 1) :: (♦ , 10) :: (■ , 100) :: []
```

could be interpreted as a function:

```
f : X → ℕ
f ▲ = 1
f ♦ = 10
f ■ = 100
```

But not any list `xys` of type `List (X × Y)` can be turned into a function. We need two additional properties:

1. For the function `f` to be total, each element of the domain must appear in `xys` paired with some element of codomain. Formally, we require `All X (map proj₁ xys)`.
2. For unambiguous interpretation, the list `xys` must not contain multiple pairs with the same domain element. Formally, `NoDup (map proj₁ xys)`

For example:

```
bad1 = (▲ , 1) :: (♦ , 10) :: []
bad2 = (▲ , 1) :: (♦ , 10) :: (♦ , 0) :: []
```

The list `bad1` violates the first requirement and the list `bad2` violates both.

Now, we translate the above into Agda:

```
Tbl : Set → Set → Set
Tbl X Y = Σ[ xys ∈ List (X × Y) ]
          All X (map proj₁ xys) ×
          NoDup (map proj₁ xys)
```

An element of type `Tbl X Y` is a list of pairs of type `X × Y` with some additional information, namely, proofs that the list of pairs is complete and duplicate-free regarding the first components. Recall that `All X xs` implies `Listable X`.

Since, for small tables, proofs of `All X` and `NoDup` can be inferred by the type-checker, it makes sense to define the following function for creating tables:

```
lstbl2dall : {X : Set} → Listable X
     → (xs : List X) → Dec (All X xs)

lstbl2dnd : {X : Set} → Listable X
     → (xs : List X) → Dec (NoDup xs)

createTbl : {X Y : Set} → (p : Listable X)
  → (xys : List (X × Y))
  → {‖ lstbl2dall p (map proj₁ xys) ‖}
  → {‖ lstbl2dnd  p (map proj₁ xys) ‖}
  → Tbl X Y
```

If the list `map proj₁ xys` contains all the elements of type `X` and is without duplicates, then the implicit arguments need not be supplied manually, since their types will be evaluated to $\top$ by the type-checker, so that `tt` is the only possible value.

Next, we implement a function for tabulating functions from a listable set:

```
toTbl : {X Y : Set} → Listable X
  → (X → Y) → Tbl X Y
```

Likewise, tables are convertible into functions:

```
fromTbl : {X Y : Set} → Tbl X Y → X → Y
```

We also show that converting back and forth between the two representations of the function is harmless:

```
fromto : {X Y : Set}
  → (p : Listable X)
  → (f : X → Y)
  → (x : X)
  → fromTbl (toTbl p f) x ≡ f x
```

As a final example of this subsection, we write a conversion function from `Elem MyPauli` to `Pauli`:

```
_↦_ : {U Y : Set}{eq : DecEq U}{b : Bool}
       → {D : FinSubDesc U eq b}
       → (x : U)
       → {‖ x ∈? toList D ‖}
       → Y → (Elem D × Y)
```

```
toPauli : Elem MyPauli → Pauli
toPauli = fromTbl (createTbl (lstblElem MyPauli)
  ('X' ↦ X  ::
   'Y' ↦ Y  ::
   'Z' ↦ Z  ::
   'I' ↦ I  :: []))
```

6.2 Predicate Matching

Assume that X is some finite set. How to implement in Agda a function $f : X \to Y$ that is defined piecewise:

$$f(x) = \begin{cases} f_1(x) & \text{if } p_1(x) \\ f_2(x) & \text{if } p_2(x) \\ \dots & \\ f_n(x) & \text{if } p_n(x) \end{cases}$$

One possibility is to provide an explicit table as described in the previous section. Unfortunately, if X is large this approach requires a lot of manual work. Another possibility is to encode it directly by nesting `if_then_else_` expressions:

```
f x = if p₁ x then f₁ x
        else if p₂ x then f₂ x
            else if p₃ x then f₃ x
               else ...
```

This approach is more concise than giving an explicit table, but it suffers from several of drawbacks:

1. There is always the last `else` branch, which plays the role of a "default" case. It will be applied to all elements which do not satisfy the predicates $\mathtt{P}_1 \dots \mathtt{P}_n$. The "default" branch makes it difficult to discover that some case was forgotten by mistake.
2. There is no good way of checking that the predicates cover all elements of the finite set (i.e., that no elements in the domain reach the "default" branch).
3. Also it is difficult to find whether there are perhaps some "dead" branches which are not satisfied by any element of `X`.

In what follows, we address these issues and introduce a notion of predicate matching.

We start by implementing a function `unreached` that takes a list of predicates and a list of elements and returns the list of those predicates that are not satisfied by any element:

```
unreached : {X : Set} → List (X → Bool)
  → List X → List (X → Bool)
unreached [] xs = []
unreached (p :: ps) xs =
  if (any p xs) then rc else (p :: rc)
  where
    rc = unreached ps (filter (not ∘ p) xs)
```

It is important to note that at the recursive call we filter out the elements that are satisfied by the head of the list of predicates. It means that the order of predicates matters. A predicate is only reached, if there is at least one element that satisfies it but does not satisfy any preceding predicates. We formalize this in the following soundness theorem:

```
unreachedSound : {X : Set}
  → (ps₁ ps₂ : List (X → Bool))
  → (p : X → Bool)
  → (xs : List X)
  → unreached (ps₁ ++ p :: ps₂) xs ≡ []
  → Σ[ x ∈ X ] x ∈ xs × p x ≡ true ×
  ((p' : X → Bool) → p' ∈ ps₁ → p' x ≡ false)
```

Soundness states that, if for some list `xs`, the list of predicates `ps` contains no unreachable ones, then for any split of `ps` into three parts, $\mathtt{ps} \equiv \mathtt{ps}_1 \mathbin{+\!\!+} \mathtt{p} :: \mathtt{ps}_2$, there exists at least one element `x` that satisfies `p` but does not satisfy any of the predicates in $\mathtt{ps}_1$.

On the other hand, completeness states that, if there exists an element `x` of the list `xs` that does not satisfy any of the predicates in `ps` and does satisfy some predicate `p`, then the list `ps ++ p :: []` is also reachable:

```
unreachedComplete : {X : Set}
  → (ps : List (X → Bool))
  → (xs : List X)
  → unreached ps xs ≡ []
  → (p : X → Bool)
  → (x : X)
  → x ∈ xs
  → p x ≡ true
  → ((p' : X → Bool) → p' ∈ ps → p' x ≡ false)
  → unreached (ps ++ p :: []) xs ≡ []
```

Now, let us address the issue of unmatched elements. We implement a function `unmatched` that returns the list of all those elements in a given list `xs` that do not satisfy any predicate in the given list `ps`:

```
isMatched : {X : Set} → List (X → Bool) →  X
  → Bool
isMatched ps x = any (λ p → p x) ps


unmatched : {X : Set} → List (X → Bool) → List X
  → List X
unmatched ps [] = []
unmatched ps (x :: xs) = if (isMatched ps x)
                         then unmatched ps xs
                         else x :: unmatched ps xs
```

The soundness theorem for the `unmatched` function states that, if there are no unmatched elements in the list `xs`, then, for any element `x` in `xs`, the list of predicates `ps` can be split into three parts, $\mathtt{ps} \equiv \mathtt{ps}_1 \mathbin{+\!\!+} \mathtt{p} :: \mathtt{ps}_2$, so that no predicate from $\mathtt{ps}_1$ is satisfied by `x` and `p` is satisfied by `x`:

```
unmatchedSound : {X : Set}
  → (ps : List (X → Bool))
  → (xs : List X)
  → unmatched ps xs  ≡ []
  → (x : X) → x ∈ xs
  → Σ[ ps₁ ∈ List (X → Bool) ]
    Σ[ ps₂ ∈ List (X → Bool) ]
    Σ[ p ∈ (X → Bool) ]
    ps₁ ++ p :: ps₂ ≡ ps ×
    isMatched ps₁ x ≡ false ×
    p x ≡ true
```

Completeness says that, if each element in the list `xs` satisfies at least one predicate in `ps`, then there are no unmatched elements:

```
unmatchedComplete : {X : Set}
  → (ps : List (X → Bool))
  → (xs : List X)
  → ((x : X) → x ∈ xs
          → Σ[ p ∈ (X → Bool) ]
             p ∈ ps × p x ≡ true)
 → unmatched ps xs ≡ []
```

We can now define a combinator that takes a list of predicates and functions from a listable set with proofs thats all predicates are reached and all elements matched, and returns a function built from the pieces:

```
predicateMatching : {X Y : Set}
  → (ps : List ((X → Bool) × (X → Y)))
  → (p : Listable X)
  → unmatched (map proj₁ ps) (proj₁ p) ≡ []
  → unreached (map proj₁ ps) (proj₁ p) ≡ []
  → X → Y
```

Let us look at some examples, but first, we want to have a combinator `fromPure` for restricting the domain of a function to a finite subset:

```
fromPure : {U Y : Set}{eq : DecEq U}{b : Bool}
  → {D : FinSubDesc U eq b}
  → (U → Y)
  → Elem D → Y
fromPure f (x , _) = f x
```

We define a finite subset of naturals `MyNats` containing five natural numbers.

```
MyNats : FinSubDesc ℕ _≟_ false
MyNats = fsd-nodup (1 :: 42 :: 3 :: 8 :: 17 :: [])
```

Next we define a function `even2odd3` that doubles the even and triples the odd numbers of `MyNats`:

```
even2odd3 : Elem MyNats → ℕ
even2odd3 = predicateMatching
  (fromPure odd  , (λ  (x , p) → x * 3)  ::
   fromPure even , (λ  (x , p) → x * 2)  :: [])
  (lstblElem MyNats) refl refl
```

The two last arguments (`refl`) are proofs of `[] ≡ []` and indicate that there are no unmatched elements and no unreachable predicates. However, if we remove the first equation

```
even2odd3Bad1 = predicateMatching
  (fromPure even , (λ  (x , p) → x * 2)  :: [])
  (lstblElem MyNats) ??? refl
```

then there are unmatched elements and the type-checker wants us to supply a proof of

```
(1 , tt) :: (3 , tt) :: (17 , tt) :: [] ≡ []
```

for the hole. The goal gives us a nice hint about which elements exactly are unmatched.

If instead we replace the first equation with a predicate which is satisfied by any element

```
even2odd3Bad2 = predicateMatching
  ((λ _ → true)  , (λ  (x , p) → x * 3)  ::
   fromPure even , (λ  (x , p) → x * 2)  :: [])
  (lstblElem MyNats) refl ???
```

then the type-checker asks us to prove that `fromPure even :: [] ≡ []`, which again hints which equations are unreachable.

7. Prover

7.1 Motivation

The module `Data.Fin.Dec` of the standard library of Agda [3] is a toolkit for building deciders of properties of elements of `Fin n`. The library contains many combinators, but for illustration purposes, it is enough to look at one them:

```
all? : {n : ℕ} {P : Fin n → Set}
     → ((i : Fin n) → Dec (P i))
     → Dec ((i : Fin n) → P i)
```

The combinator `all?` takes some decidable predicate `P` on elements of `Fin n` and returns a decision of whether `P` holds for all elements of `Fin n`.

Suppose we want to use `all?` to establish the property from Section 1, namely, commutativity of the operation `_·_ Pauli`. To do so, we can use the previously established fact that there is a bijection `f2p` from `Fin 4` to `Pauli` and decide by using `all?` whether `f2p i₁ · f2p i₂` is equal to `f2p i₂ · f2p i₁` for all `i₁` and `i₂`:

```
commDec : Dec ((i₁ i₂ : Fin 4)
  → f2p i₁ · f2p i₂ ≡ f2p i₂ · f2p i₁)
commDec = all? (λ i₁ →
      all? (λ i₂ →
         (f2p i₁ · f2p i₂) ≡P? (f2p i₂ · f2p i₁)))
```

Then, using the proof `f2p-surj` of `p2f` being a pre-inverse of `f2p`, we can establish the property itself:

```
·-comm : (x₁ x₂ : Pauli) → x₁ · x₂ ≡ x₂ · x₁
·-comm x₁ x₂ with fromSq commDec (p2f x₁) (p2f x₂)
·-comm x₁ x₂
    | p rewrite f2p-surj x₁ | f2p-surj x₂ = p
```

This approach appears to generate much less boilerplate comparing than the direct proof given in Section 1. However, there are two shortcomings that we would like to eliminate:

1. The standard library combinators work with `Fin n`. Therefore, before setting out to prove anything about some finite type, we need to provide a bijection from an initial segment of natural numbers. In Section 3.3, we showed that for listable subsets this is not always possible.
2. The property is then first proved for `Fin n` (`commDec`) and then mapped back to `Pauli` using the conversions `f2p` and `p2f` and the proof `f2p-surj`.

7.2 Definition

We start by defining a combinator `subAll?` which is very similar to `all?` shown above:

```
subAll? : {U : Set}{P : U → Set}
  → ListableSub U P
  → {Q : U → Set}
  → ((x : U) → {P x} → Dec (Q x))
  → Dec ((x : U) → {P x} → Q x)
```

The main difference is that the predicates `P` and `Q` now range over some listable subset instead of `Fin n`. Recall that the elements of `ListableSub U P` are the elements of `U` satisfying the `P`.

The same can be done for the existential quantifier:

```
subAny? : {U : Set}{P : U → Set}
  → ListableSub U P
  → {Q : U → Set}
  → ((x : U) → {P x} → Dec (Q x))
  → Dec (Σ[ x ∈ U ] P x × Q x)
```

If `Q` is a decidable predicate on some subset, then we can find out whether at least one element of that subset satisfies `Q`.

The combinators `subAll?` and `subAny?` are sufficient to decide properties which are in prenex form with the quantifiers ranging over the whole finite subset given by `P`. However, for the convenience of the user we have also added combinators for restricted quantification. These combinators allow narrowing the range of quantification by a further predicate decidable on the subset. We will not discuss them here.

Now we can provide some syntactic sugar for our combinators:

```
syntax subAll? f (λ x → z) = Π x ∈ f , z
```

```
syntax subAny? f (λ x → z) = ∃ x ∈ f , z
```

(Agda will automatically rewrite expressions matching the right hand side into the corresponding terms on the left.)

7.3 Example

Recall that the elements of `ListableSub U P` are the elements of `U` that satisfy `P`. For the special case when `U` is a listable set and `P = λ _ → ⊤`, we have simplified versions of `subAll?` and `subAny`, eliminating the overhead of dealing with trivial proofs of `P x` when `P = λ _ → ⊤`.

The proof of commutativity of the operation `_·_` on `Pauli` amounts essentially to just restating the property:

```
·-comm : (x₁ x₂ : Pauli) → x₁ · x₂ ≡ x₂ · x₁
·-comm = fromSq (
   Π x₁ ∈ listablePauli ,
   Π x₂ ∈ listablePauli , x₁ · x₂  ≡P? x₂ · x₁ )
```

The proof that the group operation has a left unit is similar:

```
·-id : Σ[ x ∈ Pauli ] (y : Pauli) → x · y ≡ y
·-id = fromSq (
   ∃ x ∈ listablePauli ,
   Π y ∈ listablePauli , x · y  ≡P? y )
```

8. Related Work

Intuitionistic frameworks give rise to a rich variety of notions of finiteness that collapse classically. In [8], [5] and [14], the author describe various concepts of finiteness and their interrrelation. According to their classification, this paper focuses on the strongest notion of finiteness, namely, finitely enumerable (listable) sets.

Since finite sets are essential for many formal theories, the users of proof assistants are asking for ways to define new finite sets [1, 7] and the developers are implementing libraries.

The Agda standard library [3] contains a toolkit for building deciders of properties of `Fin n`. Before using it for proving the property of a finite set `X`, the user needs to provide a bijection from an initial segment of natural numbers. After establishing the property for `Fin n`, it can be lifted to the original set `X`.

In [9], Gélineau improves the approach of the standard library by implementing an elegant library in Agda for proving properties quantified over finite sets. The user of a library is only asked to prove finiteness of of the set of interest by specifying a bijection from `Fin n` and then the property can be checked without transporting it to and from `Fin n` manually.

In [15], Spiwack implements a Coq library for finite subsets of countable sets. Countable sets are sets equipped with a surjection from $\mathbb{N}$. Countable sets have decidable equality: it is sufficient to test for equality the natural numbers corresponding to the elements. Finite sets then can be specified by providing a list of elements of the countable base set without duplicates. The library has support for proving decidable propositions and has a syntax for defining sets by comprehension.

In Ssreflect [10], a finite type is a type together with an explicit enumeration of its elements. Finite types can be constructed from finite duplicate-free sequences. Finite types come with boolean quantifiers `forallb` and `existsb` taking boolean predicates and returning booleans. If `X` is a finite type, the type `{set X}` is the type of sets over `X` , which is itself a finite type. Ssreflect provides the usual set theoretic-operations including membership and set comprehensions.

The authors of [4] show a systematic way for building combinators for finite sets declaratively and provide lemmas that encapsulate commonly used reasoning steps. Their work is implemented on top of Ssreflect.

9. Conclusions

In this work we addressed the problem of programming with finite sets in the dependently typed setting of the Agda programming language.

We showed that the direct approach of defining listable types as datatypes with nullary constructors is verbose and introduces brittle interdependencies between different definitions that are tedious to maintain.

Afterwards, we introduced different variations of the notion of a listable set. We proved that giving a complete list of elements of a set is equivalent to providing a surjection from an initial segment of natural numbers. Also, giving a complete list of elements without duplicates is equivalent to providing a surjection. Moreover, all four definitions are equivalent, the reason being that equality on a listable set is decidable.

Next, we introduced a more general notion of a listable subset (where a subset is specified by a base set and a predicate, but not necessarily explicitly constructed as a set). We showed that, in general, listability of a subset does not imply decidability of equality on its elements. We also proved that the union, intersection, product, and disjoint sum of listable subsets are listable subsets.

Then we proposed a pragmatic way of specifying a finite set as a subset of an already constructed set with decidable equality. A specification in this form defines a set that is listable and as a consequence also has decidable equality.

We developed two approaches for defining functions from listable subsets. In the first approach, we convert a well-formed list of argument–value pairs into a function. This is convenient to use for smaller domains. The second approach uses a list of predicate–function pairs and proofs that the predicates cover the whole domain and there are no unreachable predicates. The user receives feedback from the type-checker about predicates that are not reached and elements of the domain that are not matched.

Finally, we implemented combinators for proving propositions quantified over listable subsets. The unusual aspect is that they can be used even for subsets without decidable equality.

Acknowledgement The first author thanks Anna and Albert for their support and patience.

This research was supported by the ERDF funded Estonian CoE project EXCS, the Estonian Ministry of Education and Research institutional research grant no. IUT33-13 and the Estonian Science Foundation grant no. 9475.

References

[1] The Agda Community. Collections/containers/finite sets. *The Agda Mailing List*, 2011. `http://comments.gmane.org/gmane.comp.lang.agda/3326`

[2] The Agda Team. *The Agda Wiki*, 2015. `http://wiki.portal.chalmers.se/agda/`

[3] The Agda Team. The Agda standard library version 0.9, 2014. `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.StandardLibrary`.

[4] Y. Bertot, G. Gonthier, S. O. Biha, I. Pasca. Canonical big operators. In O. A. Mohamed, C. A. Muñoz, S. Tahar, eds., *Proc. of 21st Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLs 2008*, v. 5170 of *Lect. Notes in Comput. Sci.*, pp. 86–101. Springer, 2008.

[5] M. Bezem, K. Nakata, T. Uustalu. On streams that are finitely red. *Log. Methods in Comput. Sci.*, v. 8, n. 4, article 4, 2012.

[6] E. Bishop, D. Bridges. *Constructive Analysis*. V. 279 of *Grundlehren der mathematischen Wissenschaften*. Springer, 1985.

[7] The Coq Community. Finite sets in proofs. *The Coq Mailing List*, 2010. `http://comments.gmane.org/gmane.science.mathematics.logic.coq.club/4682`.

[8] T. Coquand, A. Spiwack. Constructively finite? In L. Laureano Lambán, A. Romero, J. Rubio, eds., *Contribuciones científicas en honor de Mirian Andrés Gómez*, pp. 217–230. Universidad de La Rioja, 2010.

[9] S. Gélineau. Library for proving propositions quantified over finite sets, 2011. `https://github.com/agda/agda-finite-prover`

[10] G. Gonthier, A. Mahboubi, E. Tassi. A small scale reflection extension for the Coq system. Rapport de recherche RR-6455. INRIA, 2008. `http://hal.inria.fr/inria-00258384`.

[11] K. Kuratowski. Sur la notion d'ensemble fini. *Fund. Math.*, v. 1, pp. 129–131, 1920.

[12] M. Nielsen, I. Chuang *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.

[13] U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, S. D. Swierstra, eds., *Revised Lectures from 6th Int. School on Advanced Functional Programming, AFP 2008*, v. 5832 of *Lect. Notes in Comput. Sci.*, pp. 230-266. Springer, 2009.

[14] E. Parmann. Some varieties of constructive finiteness. In *Abstracts of 19th Int. Conf. on Types for Proofs and Programs*, pp. 67–69. 2014.

[15] A. Spiwack. A Coq library for extensional finite sets and comprehension, 2014. `https://github.com/aspiwack/finset`

[16] P. Wadler. Comprehending monads. *Math. Struct. in Comput. Sci.*, v. 2, n. 4, pp. 461-493, 1992.

Paper II

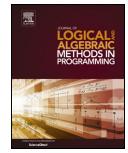
D. Firsov, T. Uustalu. **Certified CYK parsing of context-free languages.** *J. of Log. and Algebr. Meth. in Program.*, v. 83(5–6), pp. 459–468, 2014.

Journal of Logical and Algebraic Methods in Programming 83 (2014) 459–468

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Certified CYK parsing of context-free languages ☆

Denis Firsov *, Tarmo Uustalu

Institute of Cybernetics at Tallinn University of Technology, Akadeemia tee 21, 12618 Tallinn, Estonia

ARTICLE INFO

Article history:
Received 9 July 2013
Received in revised form 8 September 2014
Accepted 19 September 2014
Available online 11 October 2014

Keywords:
Certified programs
Parsing
Cocke–Younger–Kasami algorithm
Dependently typed programming
Agda

ABSTRACT

We report a work on certified parsing for context-free grammars. In our development we implement the Cocke–Younger–Kasami parsing algorithm and prove it correct using the Agda dependently typed programming language.

1. Introduction

In previous work [1], we implemented a certified parser-generator for regular languages based on the Boolean matrix representation of nondeterministic finite automata using the dependently typed programming language Agda [2,3]. Here we want to show that a similar thing can be done for a wider class of languages.

We decided to implement the Cocke–Younger–Kasami (CYK) parsing algorithm for context-free grammars [4], because of its simple and elegant structure. The original algorithm is based on multiplication of matrices over sets of nonterminals. We digress slightly from this classical approach and use matrices over sets of parse trees. By this we immediately achieve soundness of the parsing function and also eliminate the final step of parse tree reconstruction.

We first develop a simple functional version that is easily seen to be correct. We show that the memoizing version computes the same results, but without the excessive recomputation of intermediate results. The memoized version is more efficient for non-ambiguous grammars, but can be exponential for ambiguous ones since the algorithm is complete—it generates all possible parse trees.

Valiant [5] showed how to modify the CYK algorithm so as to use Boolean matrix multiplication (by encoding sets of nonterminals as binary words of some fixed length). Valiant's algorithm computes the same parsing table as the CYK algorithm, but he showed that algorithms for efficient Boolean matrix multiplication can be utilized for performing this computation, thereby achieving better worst-case time complexity. Here we refrain from pursuing this approach, since the details of the lower-level encoding obfuscate the higher-level structure of the algorithm.

We find that the main contributions of the paper are:

☆ This article is a full version of the extended abstract presented at the 24th Nordic Workshop on Programming Theory, NWPT 2012.

* Corresponding author.
E-mail addresses: denis@cs.ioc.ee (D. Firsov), tarmo@cs.ioc.ee (T. Uustalu).

http://dx.doi.org/10.1016/j.jlamp.2014.09.002
2352-2208/

- Identification of data structures that facilitate simpler proofs: the custom parsing relation, the type of certified memoization tables.
- Structuring the association lists based implementation of the algorithm using the list monad: most of the proofs rely only on the properties of the bind operation and not its implementation.
- A modular correctness proof: first, we show the correctness of a naive version of the algorithm; next, by using certified memoization tables, we lift the results to the more efficient version.

The paper is organized as follows. Section 2 presents the definitions of a context-free grammar in Chomsky normal form, parsing relations, and the naive version of the CYK algorithm. Next, in Section 3, we show that it is correct. Section 4 is devoted to discharging the obligations of the termination checker. Section 5 reports how memoization can be introduced systematically, maintaining the correctness guarantee. Since we have chosen to represent matrices as association lists, our development is heavily based on manipulation of lists and reasoning about them. Section 6 shows how we structure it using the list monad and some theorems about lists.

To avoid notational clutter, in the paper we employ an easy-to-read unofficial list comprehension syntax for monadic code instead of using monad operations directly.

Agda 2.3.3 and Agda Standard Library 0.7 were used for this development. The Agda code is available online at http://cs.ioc.ee/~denis/cert-cfg.

2. The algorithm

2.1. Context-free grammars

We will work with context-free grammars in Chomsky normal form. Rules in normal form (or more precisely, the data of such rules, i.e., allowed pairs of left and right hand sides) are given by the datatype `Rule`, which is parameterized by two types `N` and `T` for nonterminals and terminals respectively:

```
data Rule (N T : Set) : Set where
  _⟶_   : N → T → Rule N T
  _⟶_•_ : N → N → N → Rule N T
```

This definition introduces a datatype with two constructors `_⟶_` and `_⟶_•_`. The arguments of the constructors go in places of `_` in the same order as they appear in the type signature of the constructor. For example, if `A B C : N` and `a : T`, then `A ⟶ B • C` and `A ⟶ a` are inhabitants of type `Rule N T`.

A context-free grammar in Chomsky normal form is a record of type `Grammar` specifying two sets `N` and `T` for the nonterminals and terminals of the grammar, a boolean flag `nullable` indicating whether the language of the grammar contains the empty string, a nonterminal `S` for the start nonterminal, proofs that `S` never appears on the right hand side of a rule of the grammar, and finally decidable equalities on `N` and `T`.

```
record Grammar : Set₁ where
  field
    N : Set
    T : Set
    nullable : Bool
    S : N
    Rs : List (Rule N T)
    S-NT-axiom₁ : (A B : N) → (A ⟶ S • B) ∉ Rs
    S-NT-axiom₂ : (A B : N) → (A ⟶ B • S) ∉ Rs
    _=n_ : (A B : N) → (A ≡ B) ∨ (A ≢ B)
    _=t_ : (a b : T) → (a ≡ b) ∨ (a ≢ b)
```

(Note that, since two fields of the record type `Grammar` range over the type `Set`, the type `Grammar` itself cannot be a member of `Set`, but has to belong to the next universe `Set₁`.) We define two abbreviations `String = List T` and `Rules = List (Rule N T)`.

The proposition `x ∈ xs` expresses that `x` is an element of `xs`. A proof of this proposition identifies a position in the list. The rules of a grammar do not have names, they are identified by their positions in the enumeration `Rs`. A grammar can have several rules with the same left and right hand sides, only differing by their identities, i.e., positions in the list `Rs`.

Henceforth, we assume one fixed grammar `G` in the context, so we use the fields of the grammar record directly, e.g., `Rs` and `nullable` instead of `Rs G` and `nullable G` for some given `G`.

2.2. Parsing relation

Before describing the parsing algorithm, we must define correctly constructed parse trees. The most intuitive definition looks as follows:

```
data _▶_  : (s : String) → N → Set where
  empt :  nullable ≡ true → [] ▶ S
  sngl : {A : N} → {a : T} → (A ⟶ a) ∈ Rs → [ a ] ▶ A
  cons : {A B C : N} → {s₁ s₂ : String}
         → (A ⟶ B • C) ∈ Rs
         → s₁ ▶ B
         → s₂ ▶ C
         → (s₁ ++ s₂) ▶ A
```

(Arguments enclosed in curly braces are implicit. The type checker will try to figure out the argument value for you. If the type checker cannot infer an implicit argument, then it must be provided explicitly.)

The proposition `s ▶ A` states that the string `s` is derivable from nonterminal `A`. Proofs of this proposition are parse trees.

1. If `nullable` is true, then `empt` constructs a parse tree for the empty string.
2. If `A ⟶ a ∈ Rs` for some `A`, then the constructor `sngl` builds a parse tree for string `[ a ]` starting from `A`.
3. If t_1 is a parse tree starting from `B`, t_2 is a parse tree from `C` and `A ⟶ B • C ∈ Rs` for some `A`, then the constructor `cons` combines those trees into a tree starting from `A`.

However, to move closer to the structure of the CYK algorithm, we define a more specific of the parsing relation. For any given string `s` we define the parsing relation of its substrings as an inductive predicate on two naturals:

```
data _[_,_)▶_ (s : String) : ℕ → ℕ → N → Set where
  empt : {i : ℕ} → nullable ≡ true → i ≤ length s → s [ i, i )▶ S
  sngl : {i : ℕ}{A : N} → (A ⟶ charAt i s) ∈ Rs
           → s [ i, suc i )▶ A
  cons : {i j k : ℕ} → {A B C : N} → (A ⟶ B • C) ∈ Rs
           → s [ i, j )▶ B
           → s [ j, k )▶ C
           → s [ i, k )▶ A
```

The proposition `s [ i, j )▶ A` states that the substring of `s` from the `i`-th position (inclusive) to the `j`-th (exclusive) is derivable from nonterminal `A`. Proofs of this proposition are parse trees.

In particular, the string `s` is in the language of the grammar if it is derivable from `S`, i.e., we have a proof of `s [ 0, length s )▶ S`.

The only important difference between relations `_▶_` and `_[_,_)▶_` is that the second one has a fixed string `s` and constructs parse trees only for substrings of `s`, identifying them by start and end positions. Finally, we would like to define mappings between the two representations of the parse trees. To start with, we define a substring function:

```
sub : String → (i n : ℕ) →  String
sub s i n = take n (drop i s)
```

The term `sub s i n` evaluates to the substring of `s` from position `i` to position `i + n`.

Next, we prove that, for any parse tree in `s [ i, n + i )▶ A`, we can construct `(sub s i n) ▶ A`.

```
▶sound : (A : N) → (s : String) → (i n : ℕ)
                      → s [ i, n + i )▶ A  → (sub s i n) ▶ A
```

Conversely, for any tree `(sub l i n)▶ A`, an alternative tree `s [ i, n + i )▶ A` can be constructed. Note that, by the definition of `sub`, if `n + i ≥ length s`, then `sub s i n ≡ sub s i n′`, where `n′` is chosen so that `n′ + i ≡ length s`.

```
▶complete : (A : N) → (s : String) → (i n : ℕ)
     → n + i ≤ length s → (sub s i n) ▶ A  → s [ i, n + i )▶ A
```

In the rest of the paper, the parsing relation `_[_,_)▶_` will be used.

2.3. Parsing algorithm

The algorithm works with matrices of sets of parse trees where the rows and columns correspond to two positions in some string `s`. The only allowed entries at a position `i`, `j` are parse trees from various nonterminals for the substring of `s` from the `i`-th to the `j`-th position.

We represent matrices as association lists of elements of the form (row, column, entry). Note that we allow multiple entries in the same row and column of a matrix, corresponding to multiple parse trees for the same substring (and possibly the same nonterminal).

```
Mtrx : String → Set
Mtrx s = List (∃[i : ℕ] ∃[j : ℕ] ∃[A : N] s [ i, j )► A)
```

Let m_1 and m_2 be two matrices for the same string `s`. Their product is defined as:

```
_*_  : Mtrx s → Mtrx s → Mtrx s
m₁ * m₂ = { (i, k, A, cons _ t₁ t₂) | (i, j, B, t₁) ← m₁,
              (j, k, C, t₂) ← m₂, (A ⟶ B • C) ← Rs }
```

The operation `_*_` is multiplication (in the ordinary sense of matrix multiplication) of two matrices over sets of parse trees. The product of two sets of parse trees is given by the set of all well-typed parse trees `cons p` t_1 t_2 where t_1 is drawn from the first set and t_2 from the second. The sum of two sets of parse trees is their union.

Next, we define a function `triples` which, given a natural number `n`, enumerates all pairs of natural numbers which add up to `n`:

```
triples : (n : ℕ) → List (∃[i : ℕ] ∃[j : ℕ] i + j ≡ n)
triples = { (i, n - i, +-eq n i) | i ← [0 ... n] }
  where
    +-eq : (n : ℕ)(i : [0 ... n]) → i + (n - i) ≡ n
    +-eq = ...
```

For a matrix `m` we define raising `m` to the `n`-th "power" as:

```
pow : Mtrx s → ℕ → Mtrx s
pow m zero = if nullable
               then { (i, i, S, empt _) | i ← [0 ... length s) }
               else []
pow m (suc zero) = m
pow m (suc (suc n)) = { t | (i, j, _) ← triples n,
                            t ← pow m (suc i) * pow m (suc j) }
```

Let us give an example:

```
pow m 4 = m * (pow m 3) ++ (pow m 2) * (pow m 2) ++ (pow m 3) * m
pow m 3 = m * (pow m 2) ++ (pow m 2) * m
pow m 2 = m * m
```

Note that this function is not structurally recursive. The numbers `suc i`, `suc j` returned by `triples` are in fact smaller than `suc (suc n)`, but not by definition, only provably. We will introduce a structurally recursive version in Section 4.

The CYK parsing algorithm takes a string `s` and checks whether `s` can be derived from the start nonterminal `S`. Since, for any grammar in Chomsky normal form, there can only be a finite number of parse trees for any given string, our version of the algorithm faithfully returns a list of all possible derivations (trees) of string `s` from all nonterminals.

We record all parse trees of all length-1 substrings of `s` in the matrix `m-init s`:

```
m-init : (s : String) → Mtrx s
m-init s = { (i, suc i, A, sngl _) | i ← [0 ... length s),
                          (A ⟶ charAt i s) ← Rs }
```

As a result, `pow (m-init s) n` contains exactly all parse trees of all length-`n` substrings of `s`. Indeed, the intuition is as follows. The empty string is parsed, if the grammar is nullable. And any string of length 2 or longer has its parse trees given by a binary rule and parse trees for shorter strings. We give a formal correctness argument soon.

To find the parse trees of the full string `s` for the start nonterminal `S`, we compute `pow (m-init s) (length s)` and extract the parse trees for `S`.

```
cyk-parse : (s : String) → Mtrx s
cyk-parse s = pow (m-init s) (length s)

cyk-parse-main : (s : String) → List (s [ 0, length s )▶ S)
cyk-parse-main s =
    { (i, j, A, t) ← cyk-parse s, A == S, i == 0, j == length s }
```

(`cyk-parse s` can have entries only in row `0`, column `length s`, but Agda does not know this, unless we invoke the lemma `sound` below.)

3. Correctness

Correctness of the algorithm means that it defines the same parse trees as the parsing relation. We break the correctness statement down into soundness and completeness.

Soundness in the sense that `pow (m-init s) n` produces good parse trees of substrings of `s` is immediate by typing. With minimal reasoning we can also conclude

```
sound : (s : String) → (i j n : ℕ) → (A : N) → (t : s [ i, j )▶ A))
  → (i, j, A, t) ∈ pow (m-init s) n → j ≡ n + i
```

i.e., only substrings of length `n` are derived at stage `n`.

To prove completeness, we need to show that `triples n` contains all possible combinations of natural numbers `i` and `j` such that `i + j ≡ n`:

```
triples-complete : (i j n : ℕ)
     → (prf : i + j ≡ n) → (i, j, prf) ∈ triples n
```

This property is proved by induction on `n`.

It is easily proved that a proof of `s [ i, j )▶ A` with `A` not equal to `S` parses a non-empty substring of `s`.

```
compl-help : (s : String) → (i j : ℕ) → (A : N)
    → s [ i, j )▶ A → A ≢ S → ∃[n : ℕ] j ≡ suc n + i
```

Now, we are ready to show that the parsing algorithm is complete.

```
complete : (s : String) → (i n : ℕ) → (A : N)
    → (t : s [ i, n + i )▶ A) → (i, n + i, A, t) ∈ pow (m-init s) n
```

The proof is by induction on the parse tree `t`. Let us analyze the possible cases:

- If `t = empt prf` for some `prf` of type `nullable ≡ true`, then `n = 0` and the first defining equation of the function `pow` applies:

  ```
  pow (m-init s) 0 = [(i, i, S, empt _) | i ← [0 ... length s)].
  ```

  which clearly contains `t`.
- If `t = sngl p` for some `p` of type `A ⟶ charAt i s ∈ Rs`, then `n = 1` and `pow (m-init s) 1 = m-init s`. By definition, `m-init s` contains all possible derivations of single terminals found in `s`.
- In the third case, `t = cons p t₁ t₂` for some `p` of type `A ⟶ B • C ∈ Rs`, `t₁` of type `s [ i, j )▶ B`, `t₂` of type `s [ j, n + i )▶ C`.
  - If `B` or `C` are equal to `S`, then we get contradiction with `S-axiom₁` or `S-axiom₂` respectively.
  - If neither `B` or `C` is equal to `S`, then by `compl-help` we get `j ≡ suc c + i` for some `c` and `n + i ≡ suc d + suc c + i`, for some `d`. By the induction hypothesis, we get that `(i, suc c + i, B, t₁) ∈ pow (m-init s) (suc c)` and `(suc c + i, suc d + suc c + i, C, t₂)∈ pow (m-init s) (suc d)`. Hence, from the definition of multiplication and `A ⟶ B • C ∈ Rs` we conclude that

    ```
    (i, suc d + suc c + i, A, t) ∈ pow (m-init s) (suc c) *
                                    pow (m-init s) (suc d)
    ```

    From `n + i ≡ suc d + suc c + i` we get `n ≡ suc (suc (c + d))` and by `triples-complete` we know that `(c, d, refl) ∈ triples (c + d)` where `refl : c + d ≡ c + d`. Since `pow` computes and unions all the products of pairs returned by `triples (c + d)` we conclude that

```
pow (m-init s) (suc c) ∗ pow (m-init s) (suc d)
                                        ⊆  pow (m-init s) n.
```

which completes the proof.

Note that this proof of correctness makes explicit the induction principles employed and other details which are usually left implicit in textbook expositions of the algorithm.

In our Agda development, we have implemented the algorithm together with the completeness proofs just shown. The most interesting part of implementation is the design of data structures together with some useful invariants which support smooth formal proofs.

4. Termination

For the logic of Agda to be consistent, all functions must be terminating. This is statically checked by Agda's termination checker. So it is the duty of a programmer to provide sufficiently convincing arguments. This section describes the classical approach for proving termination based on well-founded relations [6].

The definition of `pow` given above is not recognized by Agda as terminating, even if it actually terminates.

The reason is that Agda accepts recursive calls on definitionally structurally smaller arguments of an inductive type. In our case, however, a call of `pow` on `suc (suc n)` leads to calls on `suc i` and `suc j` where `(i, j, prf) ∈ triples n`, i.e., to calls on provably smaller numbers (and not on, say, just `suc n` or `n`).

To make our definition acceptable not only to Agda's type-checker, but also the termination-checker, we have to explain Agda that we make recursive calls along a well-founded relation.

Classically, we can say that a relation is well-founded, if it contains no infinite descending chains. An adequate constructive version uses the notion of accessibility.

An element `x` of a set `X` is called *accessible* with respect to some relation `_≺_`, if all elements related to `x` are accessible. Crucially, this definition is to be read inductively.

```
data Acc {X : Set}(_≺_: X → X → Set)(x : X) : Set where
  acc : ((y : X) → y ≺ x → Acc _≺_ y) → Acc _≺_ x
```

A relation can be said to be *well-founded*, if all elements in the carrier set are accessible.

```
Well-founded : {X : Set}(_≺_: X → X → Set) → Set
Well-founded = (x : X) → Acc _≺_ x
```

Now we can define the less-than relation `_<_` on natural numbers:

```
data _<_ (m : ℕ) : ℕ → Set where
  <-base : m < suc m
  <-step : {n : ℕ} → m < n → m < suc n
```

And we can prove that relation `_<_` is well-founded.

```
<-wf : Well-founded _<_
```

Finally, we can summarize everything and give a definition of `pow` that is structurally recursive on the proof of accessibility, and by doing so, discharge the obligations of the termination checker:

```
pow' : {s : String} → Mtrx s → (n : ℕ) → Acc _<_ n → Mtrx s
pow' m zero accn = if nullable
                     then { (i, i, S, empt _) | i ← [0 ... length s) }
                     else []
pow' m (suc zero) accn = m
pow' m (suc (suc n)) (acc acf) = { t | (i, j, prf) ← triples n,
     t ← (pow' m (suc i) (acf (suc i) (<-lem₁ prf))) ∗
         (pow' m (suc j) (acf (suc j) (<-lem₂ prf))) }
    where
      <-lem₁ : ∀{i j n} → i + j ≡ n → suc i < suc (suc n)
      <-lem₂ : ∀{i j n} → i + j ≡ n → suc j < suc (suc n)

pow : {s : String} → Mtrx s → (n : ℕ) → Mtrx s
pow m n = pow' m n (<-wf n)
```

After changing the function pow (namely adding the accessibility argument), the correctness proofs must also be adjusted. But the changes are minimal.

5. Memoization

Our implementation of the algorithm is well-founded recursive on the less-than relation. Without memoization, it involves excessive recomputation of the matrices pow m n. To avoid recomputation of intermediate results we implement a memoized version of the pow function.

We introduce a type of memo tables. A memo table can record some powers of m as entries; we allow only valid entries.

```
MemTbl : {s : String} → Mtrx s → Set
MemTbl {s} m = (n : ℕ) → Maybe (∃[m′ : Mtrx s] m′ ≡ pow m n)
```

In our implementation, extracting elements from the memo table takes time proportional to the number of elements in it (imperative implementations could do that in constant time).

We introduce a function pow-tbl that is like pow, except that it expects to get some element tbl of MemTbl m as an argument. Instead of making recursive calls, it looks up matrices in the given memo table tbl. If the required matrix is not there, it falls back to pow. At this stage we do not worry about where to get a memo table from; we just assume that we have one given.

```
pow-tbl : {s : String} → (m : Mtrx s) → ℕ → MemTbl m → Mtrx s
pow-tbl m zero tbl = if nullable
                       then { (i, i, S, empt _) | i ← [0 ... length s) }
                       else []
pow-tbl m (suc zero)    tbl = m
pow-tbl m (suc (suc n)) tbl = { t | (i, j, _) ← triples n,
                         t ← mt (suc i) ∗ mt (suc j) }
    where
        mt n = maybe (pow m n) fst (tbl n)

        maybe : Y → (X → Y) → Maybe X → Y
        maybe y f nothing  = y
        maybe y f (just x) = f x
```

The next step is to prove that pow and pow-tbl compute propositionally equal results.

```
pow≡pow-tbl : {s : String} → (m : Mtrx s) → (n : ℕ) →
    (tbl : MemTbl m) → pow-tbl m n tbl ≡ pow m n
```

The proof is easy. Recall that the only difference between the functions pow and pow-tbl is that the function pow calls itself while function pow-tbl first tries to retrieve the result from the memo table tbl. Let us analyze the possible cases:

- If tbl n returns nothing, then mt n returns the result of pow m n.
- If tbl n returns just p, then p is a pair of a matrix m′, which becomes mt n, and a proof that m′ equals to pow m n.

Hence the functions pow and pow-tbl are extensionally equal.

Now we have to find a way to actually build memo tables with intermediate results together with the proofs that they coincide with the matrices returned by pow.

We implement a function which iteratively computes the powers pow m n of an argument matrix m, where i ≤ n ≤ i + j for given i and j, remembering all intermediate results.

```
pow-mem : {s : String} → (m : Mtrx s) → ℕ → ℕ → MemTbl m → Mtrx s
pow-mem m i zero    tbl = pow-tbl m i tbl
pow-mem m i (suc j) tbl = pow-mem m (suc i) j tbl′ where
        tbl′ p = if p == i
                   then just (pow-tbl m i tbl, pow≡pow-tbl m i tbl)
                   else tbl p)
```

The function pow-mem calls itself with ever more filled memo tables starting from lower powers. Observe how the theorem pow≡pow-tbl is now used to ensure the correctness of each new memo table tbl′.

Finally, the function for CYK parsing can be defined as follows:

```
cyk-parse-mem : (s : String) → Mtrx s
cyk-parse-mem s =
  pow-mem (m-init s) 0 (length s) (λ _ → nothing)
```

6. List monad

In the definitions above, for the sake of clarity we used informal Haskell-style list comprehension syntax. List comprehensions give a good intuition about the properties of the functions defined. Agda does not support such syntax, but we can explicate the monad structure on lists and use that to faithfully translate the comprehension syntax into Agda. The way of translating comprehensions into monadic code was described in [7].

First, we define the "bind" and "return" operations:

```
_>>=_ : {X Y : Set} → List X → (X → List Y) → List Y
_>>=_ xs f = foldr (λ x ys → f x ++ ys) [] xs

return : {X : Set} → X → List X
return x = [ x ]
```

Second, we prove the monad laws:

- Left identity:

```
left-id : {X Y : Set} →  (x : X) →  (f : X → List Y)
      → return x >>= f ≡ f x
```

- Right identity:

```
right-id : {X : Set} → (xs : List X)
      →  xs >>= return ≡ xs
```

- Associativity:

```
assoc : {X Y Z : Set} → (xs : List X) → (f : X → List Y)
  → (g : Y → List Z)
  → (xs >>= f) >>= g ≡ xs >>= (λ x → f x >>= g)
```

Finally, we can define the translation from comprehensions to monadic code:

- For the base case we have:

```
{ t | x ← xs } = xs >>= (λ x → return t)
```

- And for the step case:

```
{ t | p ← ps, q } = ps >>= (λ p → { t | q })
```

In addition to the monad laws, we prove the following theorems about _>>=_:

– The elements of lists defined by a comprehension can be traced back to where they originate from. This theorem provides a generic way for proving properties about the elements of a comprehension:

```
list-monad-th : {X Y : Set} → (xs : List X) → (f : X → List Y)
  → (y : Y) → y ∈ xs >>= f
  → ∃[x : X]  x ∈ xs × y ∈ f x
```

– We also need to use that a comprehension does not miss anything:

```
list-monad-ht : {X Y : Set} → (xs : List X) → (f : X → List Y)
  → (y : Y) → (x : X) → x ∈ xs → y ∈ f x
  → y ∈ xs >>= f
```

– If f and g are extensionally equal (i.e., propositionally equal on all arguments), then we can change one for the other, a sort of congruence property:

```
>>=cong  : ∀ {X Y : Set} → (xs : List X) → (f g : X → List Y)
  → (∀ x → f x ≡ g x) → xs >>= f ≡ xs >>= g
```

– The next property (a corollary from the associativity law) shows that "bind" is distributive over concatenation:

```
>>=distr : {X Y : Set} → (xs ys : List X) → (f : X → List Y)
  → (xs ++ ys) >>= f ≡ (xs >>= f) ++ (ys >>= f)
```

The main proofs in our work reason about list comprehensions only with the monad laws and properties of the "bind" operator like those just outlined. This makes them modular and concise.

7. Related work

Formal verification of parsers has proven to be an interesting and challenging topic for developers of certified software.

Barthwal and Norrish [8] formalize SLR parsing using the HOL4 proof assistant. They construct an SLR parser for context-free grammars, and prove it to be sound and complete. Formalization of the SLR parser is done in over 20 000 lines of code, which is a rather big development. However, SLR parsers handle only unambiguous grammars (SLR grammars are a subset of $LR(1)$ grammars).

Parsing Expression Grammars (PEGs) are a relatively recent formalism for specifying recursive descent parsers. Koprowski and Binsztok [9] formalize the semantics of PEGs in Coq. They check context-free grammars for well-formedness. Well-formedness ensures that the grammar is not left-recursive. Under this assumption, they prove that a non-memoizing interpreter is terminating. Soundness and completeness of the interpreter are shown easily, because the PEG interpreter is a functional representation of the semantics of PEGs.

An $LR(1)$ parser is a finite-state automaton, equipped with a stack, which uses a combination of its current state and one lookahead symbol in order to determine which action to perform next. Jourdan, Pottier and Leroy [10] present a validator which, when applied to a context-free grammar G and an automaton A, checks that A and G agree. The validation process is independent of which technique was used to construct A. The validator is implemented and proved to be sound and complete using the Coq proof assistant. However, there is no guarantee of termination of interpreter that executes the $LR(1)$ automaton. Termination is ensured by supplying some large constant (fuel) to the interpreter.

Danielsson and Norell [11] implement a library of parser combinators with termination guarantees in Agda [3]. They represent parsers using clever combination of induction and coinduction which guarantees termination and also rules out some forms of left recursion.

Sjöblom [12] implements Valiant's algorithm in Agda. He shows that a context-free grammar induces a nonstandard algebraic structure—a nonassociative semiring. It is defined as a tuple $(R, 0, +, \cdot)$, where $(R, 0, +)$ is a monoid, $\cdot$ is distributive over 0 and $+$. He also defines inductive datatypes for vectors, matrices and triangular matrices. The type for triangular matrices is built of the pieces needed for recursive calls of Valiant's parsing algorithm. Finally, he shows that, given some triangular matrix over some nonassociative semiring, the algorithm implemented computes the transitive closure of the input triangular matrix.

None of the previously mentioned works can treat all context-free grammars. Ridge [13] demonstrates how to construct sound and complete parser implementations directly from grammar specifications, for all context-free grammars, based on combinator parsing. He constructs a generic parser generator and shows that generated parsers are sound and complete. The formal proofs are mechanized using the HOL4 theorem prover. The time complexity of the memoized version of the implemented parser is $O(n^5)$.

8. Conclusion and future work

Verified implementation of well known algorithms is important mainly because it encourages finding implementations that make it feasible to conduct small and elegant proofs and also makes all necessary assumptions explicit.

We have shown that, with careful design, programming with dependent types is a powerful tool for implementing algorithms together with correctness proofs.

Since the CYK algorithm handles only grammars in normal form, we plan to extend our work to grammars in general form. One possible way of doing it is to implement a verified normalization algorithm for context-free grammars, i.e., convert context-free grammars from general form to normal form. In the constructive setting, proofs of soundness and completeness of this procedure will be functions between parse trees in the general and normal-form grammars. So one could use the CYK implementation of this paper to produce parse trees for grammars in normal form and then convert them to trees for grammars in general form by using the soundness proof of the normalization algorithm.

Acknowledgements

The authors were supported by the ERDF funded Estonian CoE project no. 3.2.0101.08-0013 (EXCS), the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12 and the Estonian Science Foundation grant no. 9475.

References

[1] D. Firsov, T. Uustalu, Certified parsing of regular languages, in: G. Gonthier, M. Norrish (Eds.), Proc. of 3rd Int. Conf. on Certified Programs and Proofs, CPP 2013, in: Lect. Notes Comput. Sci., vol. 8307, Springer, Berlin, 2013, pp. 98–113.
[2] U. Norell, Towards a practical programming language based on dependent type theory, Ph.D. thesis, Chalmers University of Technology, Göteborg, 2007.
[3] U. Norell, Dependently typed programming in Agda, in: P. Koopman, R. Plasmeijer, S.D. Swierstra (Eds.), Revised Lectures from 6th Int. School on Advanced Functional Programming, AFP 2009, in: Lect. Notes Comput. Sci., vol. 5832, Springer, Berlin, 2009, pp. 230–266.
[4] D. Younger, Recognition and parsing of context-free languages in time $O(n^3)$, Inf. Comput. 10 (2) (1967) 189–208, http://dx.doi.org/10.1016/s0019-9958(67)80007-X.
[5] L.G. Valiant, General context-free recognition in less than cubic time, J. Comput. Syst. Sci. 10 (2) (1975) 308–314, http://dx.doi.org/10.1016/s0022-0000(75)80046-8.
[6] B. Nordström, Terminating general recursion, BIT Numer. Math. 28 (3) (1988) 605–619, http://dx.doi.org/10.1007/bf01941137.
[7] P. Wadler, Comprehending monads, in: Proc. of 1990 ACM Conf. on LISP and Functional Programming, LFP '90, ACM, New York, 1990, pp. 61–78.
[8] A. Barthwal, M. Norrish, Verified, executable parsing, in: G. Castagna (Ed.), Proc. of 18th Europ. Symp. on Programming Languages and Systems, ESOP '09, in: Lect. Notes Comput. Sci., vol. 5502, Springer, Berlin, 2009, pp. 160–174.
[9] A. Koprowski, H. Binsztok, TRX: a formally verified parser interpreter, Log. Methods Comput. Sci. 7 (2) (2011) art. no. 18, http://dx.doi.org/10.2168/lmcs-7(2:18)2011.
[10] J.-H. Jourdan, F. Pottier, X. Leroy, Validating LR(1) parsers, in: H. Seidl (Ed.), Proc. of 21st Europ. Symp. on Programming, ESOP 2012, in: Lect. Notes Comput. Sci., vol. 7211, Springer, Berlin, 2012, pp. 397–416.
[11] N.A. Danielsson, Total parser combinators, in: Proc. of 15th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '10, ACM, New York, 2010, pp. 285–296.
[12] T.B. Sjöblom, An Agda proof of the correctness of Valiant's algorithm for context free parsing, Master's thesis, Chalmers University of Technology, Göteborg, 2013.
[13] T. Ridge, Simple, functional, sound and complete parsing for all context-free grammars, in: J.-P. Jouannaud, Z. Shao (Eds.), Proc. of 1st Int. Conf. on Certified Programs and Proofs, CPP 2011, in: Lect. Notes Comput. Sci., vol. 7086, Springer, Berlin, 2011, pp. 103–118.

Paper III

D. Firsov, T. Uustalu. **Certified normalization of context-free grammars.** In *Proc. of 4th ACM SIGPLAN Conf. on Certified Programs and Proofs, CPP '15 (Mumbai, Jan. 2015)*, pp. 167–174. ACM Press, 2015.

Certified Normalization of Context-Free Grammars

Denis Firsov Tarmo Uustalu
Institute of Cybernetics at TUT
{denis,tarmo}@cs.ioc.ee

Abstract

Every context-free grammar can be transformed into an equivalent one in the Chomsky normal form by a sequence of four transformations. In this work on formalization of language theory, we prove formally in the Agda dependently typed programming language that each of these transformations is correct in the sense of making progress toward normality and preserving the language of the given grammar. Also, we show that the right sequence of these transformations leads to a grammar in the Chomsky normal form (since each next transformation preserves the normality properties established by the previous ones) that accepts the same language as the given grammar. As we work in a constructive setting, soundness and completeness proofs are functions converting between parse trees in the normalized and original grammars.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—correctness proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems

Keywords certified programs; context-free grammars; Chomsky normal form; normalization; dependently typed programming; Agda

1. Introduction

In formal language theory, a context-free grammar (CFG) is said to be in the Chomsky normal form (CNF), if all of its production rules are of the form: $\mathtt{A} \longrightarrow \mathtt{BC}$, $\mathtt{A} \longrightarrow \mathtt{a}$, or $\mathtt{S} \longrightarrow \varepsilon$, where A, B and C are nonterminals, a is a terminal, S is the start nonterminal. Also, neither B nor C may be the start nonterminal.

Context-free grammars in the Chomsky normal form are very convenient to work with. It is often assumed that either CFGs are given in CNF from the beginning or there is an intermediate step of normalization. For example, Minamide [8] has implemented and proved correct three sophisticated decision procedures for context-free languages specified by CNF grammars:

- inclusion between a context-free language and a regular language;
- balancedness of a context-free language;
- inclusion between a context-free language and a regular hedge language.

Having a certified implementation of normalization for CFGs enables us to lift these decision procedures to context-free languages defined by CFGs in general form without losing the guarantees of correctness.

Another example is our previous work [3], where we reported on a certified implementation of the Cocke–Younger–Kasami (CYK) parsing algorithm in the Agda dependently typed programming language [9]. The CYK algorithm works only with grammars in the Chomsky normal form. Now, with a certified implementation of the CFG normalization algorithm we extend the reach of this work. Namely, to parse a string s for some general CFG G we could proceed as follows:

- normalize G into a CNF G';
- parse s by using the certified implementation of the CYK algorithm and get a parse tree t for the grammar G';
- finally, convert the parse tree t for the grammar G' to a parse tree t' for the grammar G with the constructive soundness proof of normalization of G (which is a function from parse trees to parse trees).

Both examples demonstrate how certified normalization enables us to adopt certified development from CNF grammars to general CFGs retaining the correctness guarantees.

The full normalization transformation for a CFG is the composition of the following constituent transformations [1]:

1. elimination of all *ε-rules* (i.e., rules of the form $\mathtt{A} \longrightarrow \varepsilon$) (Section 3);
2. elimination all *unit rules* (i.e., rules of the form $\mathtt{A} \longrightarrow \mathtt{B}$) (Section 4);
3. replacing all rules $\mathtt{A} \longrightarrow \mathtt{X_1X_2} \ \ldots \ \mathtt{X_k}$ where $\mathtt{k} \geq 3$ with rules $\mathtt{A} \longrightarrow \mathtt{X_1A_1}$, $\mathtt{A_1} \longrightarrow \mathtt{X_2A_2}$, $\mathtt{A_{k\text{-}2}} \longrightarrow \mathtt{X_{k\text{-}1}X_k}$ where $\mathtt{A_i}$ are "fresh" nonterminals (Section 5.1);
4. for each terminal a, adding a new rule $\mathtt{A} \longrightarrow \mathtt{a}$ where A is a fresh nonterminal and replacing a in the right-hand sides of all rules with length at least two with A (Section 5.2).

The algorithms for the first, third and fourth transformations are functional versions of the classical imperative algorithms described, e.g., in [1]. The approach to eliminating unit rules is a little different and is designed to support certified development (uses a recursion that is easily presented as wellfounded).

We prove the correctness of this normalization transformation by showing that a given CFG and the corresponding CNF grammar accept the same language. Because we work in a constructive framework, the proof consists of total functions converting parse trees of the normalized grammar to the given grammar (soundness) and in the converse direction (completeness) (Section 6).

CPP '15, January 13–14, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676724.2693177

We used Agda 2.4.2 and Agda Standard Library 0.8.1 for this development. The full Agda code of this paper can be found at `http://cs.ioc.ee/~denis/cert-norm/`.

2. Setup

We assume that `N` and `T` are some fixed types for nonterminals and terminals respectively. We only require `N` and `T` to have decidable equality. Symbols are terminals and nonterminals. A rule is defined as a pair of a nonterminal and a list of symbols. We also define some handy abbreviations:

```
data Symbol : Set where
  nt : N → Symbol
  tm : T → Symbol

RHS = List Symbol

data Rule : Set where
  _⟶_ : N → RHS → Rule

Rules = List Rule

Ts : Rules → List T
Ts Rs = { a | A ⟶ rhs ∈ Rs, tm a ∈ rhs }

NTs : Rules → List N
NTs Rs = { A | A ⟶ rhs ∈ Rs } ∪
         { B | A ⟶ rhs ∈ Rs, nt B ∈ rhs }

String = List T
```

(To avoid notational clutter, in the paper we employ an easy-to-read unofficial list comprehension syntax.)

For now and for most of the paper, we assume that a grammar is just a list of rules, we do not assume a fixed start nonterminal. In Section 6.2, we define a grammar as a list of rules together with a designated start nonterminal.

The datatype of the parse trees (abstract syntax trees) is parametrized by a grammar `Rs` and is defined inductively as follows:

```
mutual
  data Tree (Rs : Rules) : N → String → Set where
    node : {A : N}{rhs : RHS}{s : String}
         → A ⟶ rhs ∈ Rs
         → Forest Rs rhs s → Tree Rs A s

  data Forest (Rs : Rules) :
                RHS → String → Set where
    empty  : Forest Rs [] []
    _::ₜ_  : {rhs : RHS}{s : String}
         → (t : T) →  Forest Rs rhs s
         → Forest Rs (tm t :: rhs) (t :: s)
    _::ₙ_  : {rhs : RHS}{s₁ s₂ : String}{A : N}
         → Tree Rs A s₁ → Forest Rs rhs s₂
         → Forest Rs (nt A :: rhs) (s₁ ++ s₂)
```

(In Agda, an argument enclosed in curly braces is implicit. The Agda type checker will try to figure it out. If an argument cannot be inferred, it must be provided explicitly.)

In general, the type `Tree Rs A s` collects all parse trees for a string `s` for a grammar `Rs` and a nonterminal `A` at the root. The auxiliary type `Forest Rs rhs s` collects all parse forests for a string `s` whose constituent individual parse trees are rooted at the symbols in `rhs`.

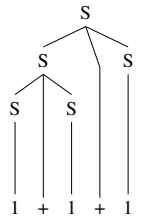
Let us look at the following example. Consider the following grammar `Rs` with two rules. Their proofs of membership in the grammar serve as names for these rules.

```
Rs : Rules
Rs = [ S ⟶ [ nt S , tm '+' , nt S ],
       S ⟶ [ tm '1' ] ]

fr : S ⟶ [ nt S , tm '+' , nt S ] ∈ Rs
sr : S ⟶ [ tm '1' ] ∈ Rs
```

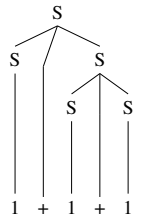
The strings `"1"` and `"1+1"` have the following unique derivations:

```
1T : Tree Rs S "1"
1T = node sr ('1' ::ₜ empty)

1+1T : Tree Rs S "1+1"
1+1T = node fr (1T ::ₙ '+' ::ₜ 1T ::ₙ empty)
```

But the string `"1+1+1"` has two derivations:




```
lft : Tree Rs S "1+1+1"
lft = node fr (1+1T ::ₙ '+' ::ₜ 1T ::ₙ empty)
```




```
rgt : Tree Rs S "1+1+1"
rgt = node fr (1T ::ₙ '+' ::ₜ 1+1T ::ₙ empty)
```

3. ε-rule elimination and its correctness

The main consequence of the presence of ε-rules in a grammar is that parse trees for the empty string can be constructed for some nonterminals. A nonterminal `A` is called *nullable* for a grammar `Rs`, if one can construct a parse tree for the empty string with `A` at the root, i.e., an inhabitant of the type `Tree Rs A []`. We describe the transformation of ε-rule elimination:

1. find all nullable nonterminals;
2. for each rule with some nullable nonterminals in its right-hand side `rhs`, add a set of new rules given by all subsequences of `rhs` obtained by dropping some nullable nonterminals;
3. remove every rule whose right-hand side is empty string.

For example, for the grammar

```
S ⟶ AbA | B
B ⟶ b | c
A ⟶ ε | d
```

the transformation produces the following grammar:

```
S ⟶ AbA | Ab | bA | b | B
B ⟶ b | c
A ⟶ d
```

Note that the transformation makes all nonterminals non-nullable: for a nonterminal nullable for the given grammar, the language of this nonterminal in the transformed grammar differs from its language in the original grammar by the absence of the empty word.

3.1 Nullable nonterminals

In this section, we describe how to find all nullable nonterminals of a grammar. We use the following observation: if a nonterminal `A` is nullable, then there exists a rule `A ⟶ rhs ∈ Rs` such that `rhs` consists only of nullable nonterminals (in particular, it is also possible that `rhs ≡ []`). Therefore, to find all nullable nonterminals, we iteratively build all trees for the empty string. Here is the algorithm:

```
nlbls : Rules → ℕ → List N
nlbls Rs zero    = start
nlbls Rs (suc n) = collect (nlbls Rs n)
  where
   start = { A | A ⟶ [] ∈ Rs }
   collect ans
    = { A | A ⟶ rhs ∈ Rs ,
                (B : N) → nt B ∈ rhs → B ∈ ans }
```

Clearly, the algorithm is sound (by construction):

```
nlbls-snd : (Rs : Rules) → (A : N) → (n : ℕ)
            → A ∈ nlbls Rs n → Tree Rs A []
```

But how many iterations do we need for the completeness? Let us look at a weak version of completeness:

```
nlbls-cmplt-weak : (Rs : Rules) → (A : N)
  → (t : Tree Rs A []) → A ∈ nlbls Rs (height t)
```

By induction on the height of the parse tree, we can easily prove this lemma. But the lemma is too weak, because it depends on the height of the input parse tree and this is not bounded. We need to find a number of iterations that is sufficient for every possible parse tree.

We prove that `length Rs` (denotes the number of the rules in the grammar) many iterations is enough:

```
nlbls-cmplt : (Rs : Rules) → (A : N)
  → Tree Rs A [] → A ∈ nlbls Rs (length Rs)
```

Proof If `height t` $\leq$ `length Rs`, then the theorem is proved by `nlbls-cmplt-weak`. If `height t` $>$ `length Rs`, then there exists at least one branch in the parse tree with at least one rule used twice. Suppose this rule is `r : B ⟶ rhs ∈ Rs`. Next, let the subtrees rooted at the left-hand nonterminal `B` of the rule `r` be `t` and `t'`, `t'` being a subtree of `t`. Both `t` and `t'` have type `Tree Rs B []`. Therefore, we can substitute `t'` for `t` and still get a parse tree of type `Tree Rs A []`. This procedure can be repeated until `height t` $\leq$ `length Rs`.

Finally, we define an abbreviation:

```
nullables : Rules → List N
nullables Rs = nlbls Rs (length Rs)
```

3.2 Subsequences

In this section, we describe how to compute certain subsequences of a list. More precisely, given some list `xs : List X` and some predicate `P : X → Bool`, we would like to compute all subsequences of `xs` obtainable by dropping some elements satisfying `P`.

```
allSubSeq : {X : Set} → (X → Bool)
                      → List X → List (List X)
allSubSeq P xs
  = foldr (λ x res →
           if P x then res ++ (map (_::_ x) res)
                  else map (_::_ x) res)
           [ [] ] xs
```

For an explanation, let us look at the example:

```
allSubSeq (≡ nt A) [ nt A, nt B, nt A, nt C ] ⇒
  [ [ nt A, nt B, nt A, nt C ],
    [       nt B, nt A, nt C ],
    [ nt A, nt B,       nt C ],
    [       nt B,       nt C ] ]
```

To generate all subsequences of some list `xs`, one could call `allSubSeq (λ _ → true) xs`. The function `allSubSeq` function is sound and complete in the sense that it generates all desired subsequences and nothing else (a formalization can be found in our development).

3.3 ε-rule elimination

Finally, to eliminate ε-rules, we combine the `allSubSeq` and `nullables`:

```
norm-e : Rules → Rules
norm-e Rs = { A ⟶ rhs' | A ⟶ rhs ∈ Rs ,
    rhs' ∈ allSubSeq (∈ nullables Rs) rhs ,
    rhs' ≢ [] }
```

First, we find all nullable nonterminals in the grammar. Then, for each rule `A ⟶ rhs` in `Rs`, we compute all subsequences `rhs'` of `rhs` obtainable by dropping some nullable nonterminals in `rhs`. Finally, for all nonempty `rhs'`, the rule `A ⟶ rhs'` is added to the resulting grammar.

3.4 Correctness

Progress Observe that the function `norm-e` explicitly excludes rules with empty right-hand sides. Therefore, it is simple to show that, for any grammar `Rs`, the normalized grammar `norm-e Rs` has no ε-rules:

```
ne-progress : (Rs : Rules) → (A : N)
                           → A ⟶ [] ∉ norm-e Rs
```

Soundness Next, let us show soundness. Namely, given some tree in the normalized grammar `Tree (norm-e Rs) A s`, we would like to construct a parse tree of `s` in the original grammar `Rs`:

```
ne-snd : (Rs : Rules) → (A : N) → (s : String)
           → Tree (norm-e Rs) A s → Tree Rs A s
```

Proof The proof is by induction on the height of `t : Tree (norm-e Rs) A s`. Pattern matching yields `f : Forest (norm-e Rs) rhs s` and `r : A ⟶ rhs ∈ norm-e Rs` such that `t ≡ node r f`. For each tree `t'` of type `Tree (norm-e Rs) B s'` such that `t' ∈ f`, by the induction hypothesis, we construct a tree `Tree Rs B s'`. Hence, we can construct `f' : Forest Rs rhs s`. Next, analyze the rule `A ⟶ rhs`. If `r' : A ⟶ rhs ∈ Rs`, then the proof is completed by the witness `node r' f'`. If `A ⟶ rhs ∉ Rs`, then by the definition of `norm-e` there exists some rule `r' : A ⟶ rhs' ∈ Rs` such that `rhs ∈ allSubSeq (∈ nullables Rs) rhs'`. By soundness of `allSubSeq`, the list `rhs` is a subsequence of `rhs'`. Moreover, the nonterminals of all removed positions in `rhs'` are contained in `nullables Rs`. Therefore, the proof can be completed by the witness `node r' f'' : Tree Rs A s`, where `f''` is constructed

from `f'` by putting trees for the empty string (produced by using soundness of `nullables`) at the dropped positions of `rhs'`.

Completeness Conversely, given a parse tree for some non-empty string (recall that `norm-e` makes all nonterminals non-nullable) in the original grammar, we can convert it into a parse tree in the normalized grammar:

```
ne-cmplt : (Rs : Rules) → (A : N) → (s : String)
  → Tree Rs A s → s ≢ [] → Tree (norm-e Rs) A s
```

Proof The proof is by induction on the height of the parse tree `t : Tree Rs A s`. By pattern matching, we have `t ≡ node r f` where `f : Forest Rs rhs s` and `r : A ⟶ rhs ∈ Rs`. For each tree `t' : Tree Rs B s'` such that `t' ∈ f`, let us analyze the possible cases:

- If `s' ≢ []`, then by the induction hypothesis, we can construct `Tree (norm-e Rs) A s'`.
- If `s' ≡ []`, then by `nlbls-cmplt`, we have that `B ∈ nullables Rs`.

Therefore, `f' : Forest (norm-e Rs) rhs' s` can be constructed where `rhs'` is a subsequence of `rhs` (the positions at which `f : Forest Rs rhs s` contains trees for the empty string are skipped). If `rhs' ≢ []`, then by completeness of `nullables` and `allSubSeq`, we get that `r' : A ⟶ rhs' ∈ norm-e Rs` and the proof is completed by the witness `node r' f'`. If `rhs' ≡ []`, then `s` should be empty (all positions are nulled), but this contradicts the assumption that `s ≢ []`.

3.5 Example

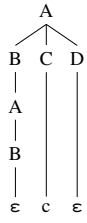
Consider the following grammar `Rs`:

```
A ⟶ BCD | B
B ⟶ ε | A
C ⟶ c
D ⟶ ε
```

Since `B` and `D` are the only nullable nonterminals, the grammar `norm-e Rs` has the following rules:

```
A ⟶ BCD | B | CD | BC | C
B ⟶ A
C ⟶ c
```

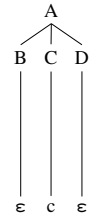
The nonterminal `D` in the grammar `norm-e Rs` is *nonproductive* (i.e., `(s : String) → Tree (norm-e Rs) D s → ⊥`). Let us look at the example of a tree `t` for the original grammar `Rs` and its counterpart for the ε-normalized grammar `norm-e Rs`:



`t : Tree Rs A "c"`

`ne-cmplt t : Tree (norm-e Rs) A "c"`

Note, that for the converse direction `ne-snd (ne-cmplt t) ≢ t` (there are many ways to construct subtrees for empty word):



`ne-snd (ne-cmplt t) : Tree Rs A "c"`

4. Unit rule elimination and its correctness

4.1 Implementation

We describe in list comprehension notation how unit rules with a particular right-hand nonterminal are eliminated:

```
nu-step : Rules → N → Rules
nu-step Rs A
 = { rule' | rule ∈ Rs, rule' ∈ step-f Rs A rule }
  where
    step-f : Rules → N → Rule → Rules
    step-f Rs A (B ⟶ rhs) =
       if rhs ≡ [ nt A ] then
        { B ⟶ rhs' | A ⟶ rhs' ∈ Rs,
                     rhs' ≢ [ nt A ] }
       else [ B ⟶ rhs ]
```

Compared to the grammar `Rs`, in the grammar `nu-step Rs A` every rule of the form `B ⟶ [ nt A ]` is replaced with all rules of the form `B ⟶ rhs'`, where `rhs'` stands for a right-hand side such that `A ⟶ rhs' ∈ Rs` and `rhs' ≢ [ nt A ]`. Now, full unit rule elimination is achieved by applying this procedure to all nonterminals:

```
norm-u : Rules → Rules
norm-u Rs = foldl nu-step Rs (NTs Rs)
```

Recall that `NTs Rs` is an enumeration of all nonterminals appearing in the grammar `Rs`.

4.2 Correctness

Progress First, we show that `nu-step` gains some progress:

```
nu-step-progress : (Rs : Rules) → (A B : N)
                 → A ⟶ [ nt B ] ∉ nu-step Rs B
```

This lemma states that there is no rule with the right-hand side `[ nt B ]` in the grammar `nu-step Rs B`. The progress lemma for the `norm-u` is a trivial consequence:

```
nu-progress : (Rs : Rules) → (A B : N)
              → A ⟶ [ nt B ] ∉ norm-u Rs
```

Soundness We start by proving a lemma about possible shapes of rules in the original grammar:

```
nu-sound-main : (Rs : Rules) → (A B : N)
  → (rhs : RHS) → A ⟶ rhs ∈ nu-step Rs B
  → A ⟶ rhs ∈ Rs
      ∨ (A ⟶ [ nt B ] ∈ Rs × B ⟶ rhs ∈ Rs)
```

This lemma shows that, if a rule `A ⟶ rhs` belongs to a normalized grammar `nu-step Rs B`, then either the rule `A ⟶ rhs` belongs to `Rs` or the rules `A ⟶ [ nt B ]` and `B ⟶ rhs` do.

Now, we show how soundness follows from `nu-sound-main`:

```
nu-step-sound : (Rs : Rules) → (A B : N)
   → (s : String)
   → Tree (nu-step Rs B) A s → Tree Rs A s
```

Proof The proof is by induction on the height of the tree `t : Tree (nu-step Rs B) A s`. Pattern matching on `t` yields some `f` of type `Forest (nu-step Rs B) rhs s` and `p : A ⟶ rhs ∈ (nu-step Rs B)` such that `t ≡ node p f`. Next, for all trees `t' : Tree (nu-step Rs B) C s'` such that `t' ∈ f`, by the induction hypothesis, we turn `t'` into `t'' : Tree Rs C s'`. Therefore, by induction on the length of `f`, a forest `f' : Forest Rs rhs s` can be constructed. Finally, by `nu-sound-main` we have two cases:

- `p' : A ⟶ rhs ∈ Rs`. Then the proof is completed by constructing the witness `node p' f' : Tree Rs A s`.
- `p' : A ⟶ [ nt B ] ∈ Rs` and `p'' : B ⟶ rhs ∈ Rs`. Then the proof is completed by the giving the witness `node ((node p'' f') ::ₙ empty) p'` which has type `Tree Rs A s`.

Soundness of `norm-u` follows trivially from `nu-step-sound`:

```
nu-snd : (Rs : Rules) → (A : N) → (s : String)
          → Tree (norm-u Rs) A s → Tree Rs A s
```

Completeness We start again by observing special properties:

```
nu-cmplt' : (Rs : Rules) → (A B : N)
  → (rhs : RHS) → A ⟶ [ nt B ] ∈ Rs
  → B ⟶ rhs ∈ Rs → rhs ≢ [ nt B ]
  → A ⟶ rhs ∈ nu-step Rs B

nu-cmplt'' : (Rs : Rules) → (A B : N)
   → (rhs : RHS) → A ⟶ rhs ∈ Rs
   → rhs ≢ [ nt B ] → A ⟶ rhs ∈ nu-step Rs B
```

The `nu-cmplt'` lemma states that, if rules `A ⟶ [ nt B ]` and `B ⟶ rhs` belong to `Rs` and `rhs ≢ [ nt B ]`, then the rule `A ⟶ rhs` belongs to the normalized grammar `nu-step Rs B`. At the same time the lemma `nu-cmplt''` establishes that rules `A ⟶ rhs` where `rhs ≢ [ nt B ]` will stay in the normalized grammar.

Using this property, completeness is proved by induction on a given parse tree and inspection of rules at two consecutive levels.

```
nu-step-complete : (Rs : Rules)
 → (A B : N) → (s : String)
 → Tree Rs A s → Tree (nu-step Rs B) A s
```

Proof The claim is proved by induction on the height of the tree `t` of type `Tree Rs A s`. Pattern matching on `t` yields some `f : Forest Rs rhs s` and `p : A ⟶ rhs ∈ Rs` such that `t ≡ node p f`. Next, for all trees `t' : Tree Rs C s'` such that `t' ∈ f`, by the induction hypothesis, we construct `t'' : Tree (nu-step Rs B) C s'`. So, by induction on the length of `f`, we get `f' : Forest (nu-step Rs B) rhs s`. Finally, let us analyze two cases:

- If `rhs ≢ [ nt B ]`, then by `nu-cmplt''` we have `p' : A ⟶ rhs ∈ nu-step Rs B` and proof is finished by the witness `node p' f' : Tree (nu-step Rs B) A s`.
- If `rhs ≡ [ nt B ]`, then the previously constructed `f'` satisfies `f' ≡ (node q f'') ::ₙ empty` where `f''` is of type `Forest (nu-step Rs B) rhs' s` and `q` is of type `B ⟶ rhs' ∈ nu-step Rs B`. By `nu-step-progress` we know that `rhs' ≢ [ nt B ]`, therefore, by `nu-cmplt'` we get `p' : A ⟶ rhs' ∈ nu-step Rs B`. Finally, the witness `node p' f'` of type `Tree (nu-step Rs B) A s` concludes the proof.

And lifting this result to the full elimination of unit rules:

```
nu-cmplt : (Rs : Rules) → (A : N) → (s : String)
        → Tree Rs A s → Tree (norm-u Rs) A s
```

4.3 Example

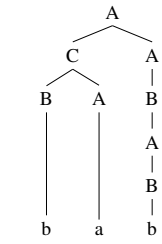
Consider the grammar

```
A ⟶ CA | B | a
B ⟶ b | A
C ⟶ BA
```

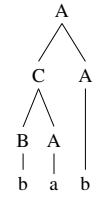
After the `norm-u` transformation we have:

```
A ⟶ CA | a | b
B ⟶ b | CA | a
C ⟶ BA
```

Observe how an example tree for the original grammar is transformed into a tree for the normalized grammar:



`t : Tree Rs A "bab"`



`nu-cmplt t : Tree (norm-u Rs) A "bab"`

Mapping this tree back from the normalized grammar to the original grammar gives a tree with the unit loop cut out:


```
nu-snd (nu-cmplt t) : Tree Rs A "bab"
```

Making the exact relationship between maps `nu-cmplt` and `nu-snd` precise is a possible future work.

5. Final transformations

5.1 Long right-hand sides

Next, we describe how to eliminate rules `A ⟶ rhs` where `length rhs > 2`—so-called *long rules*.

To do so, we first need a function that will supply fresh nonterminals,

```
newnt : Rules → N
```

and a proof that `newnt Rs` does not occur anywhere in the grammar `Rs`:

```
newnt-lem : (Rs : Rules) → newnt Rs ∉ NTs Rs
```

The above states that `newnt Rs` is a "fresh" nonterminal. Note that there are no side effects involved here, the expression `newnt Rs` always returns the same nonterminal. Hence, to get the next "fresh" nonterminal, one must first embed the current one in the grammar.

For an explanation, assume that `N = T = ℕ`. Then, let us define `newnt` and `Rs` as follows:

```
newnt : Rules → N
newnt Rs = 1 + max (NTs Rs)

Rs : Rules
Rs = [ 1 ⟶ [ nt 2, tm 3, nt 4 ] ]

Rs' : Rules
Rs' = 1 ⟶ [ nt (newnt Rs) ] :: Rs
```

In that case, `newnt Rs ≡ 5`. Also, if we define `Rs'` by adding the rule `1 ⟶ [ nt (newnt Rs) ]` to `Rs`, then `newnt Rs' ≡ 6`.

Next, we are ready to define a step of normalization:

```
nl-step' : Rules → N → Rules
nl-step' ((A ⟶ X :: Y :: Z :: rhs) :: Rs) F =
   (A ⟶ nt F :: Z :: rhs) ::
         (F ⟶ X :: Y :: []) :: Rs
nl-step' ((A ⟶ rhs) :: Rs) F =
   (A ⟶ rhs) :: nl-step' Rs F
nl-step' [] F = []

nl-step : Rules → Rules
nl-step Rs = nl-step' Rs (newnt Rs)
```

The function `nl-step` looks for the first long rule of the form `A ⟶ X :: Y :: Z :: rhs` and replaces it with rules `A ⟶ nt F :: Z :: rhs` and `F ⟶ X :: Y :: []` where `F` is fresh.

After applying the function `nl-step` to the grammar `Rs`, the sum of the lengths of the right-hand sides of all long rules decreases. This will be the measure of how many times `nl-step` needs to be applied to the grammar `Rs`.

```
nl-measure : Rules → ℕ
nl-measure Rs = sum lengths
  where
    lengths = { length rhs | A ⟶ rhs ∈ Rs,
                             length rhs > 2 }
```

So, to eliminate all long rules, we apply the function `nl-step` to the set of rules `(nl-measure Rs)` times.

```
norm-l : Rules → Rules
norm-l Rs = fold Rs nl-step (nl-measure Rs)
```

5.2 Right-hand sides containing terminals

In what follows, we describe how to eliminate rules `A ⟶ rhs` where `rhs` contains terminals and `length rhs > 1`.

The function `nt-step Rs a` adds to the grammar `Rs` the rule `newnt Rs ⟶ tm a` and substitutes the symbol `tm a` with the symbol `nt (newnt Rs)` in the right-hand side of every rule whose right-hand side is longer than 1.

```
nt-step : Rules → T → Rules
nt-step Rs a = let F = newnt Rs in
 F ⟶ tm a ::
 { A ⟶ subst (tm a) (nt F) rhs |
                          A ⟶ rhs ∈ Rs }
 where
  subst : Symbol → Symbol → RHS → RHS
  subst X Y rhs =
   if length rhs ≤ 1
   then rhs
   else map (λ Z → if Z ≡ X then Y else Z) rhs
```

Finally, remove all terminals from right-hand sides longer than 1 by folding `Rs` with the function `nt-step`:

```
norm-t : Rules → Rules
norm-t Rs = foldl nt-step Rs (Ts Rs)
```

5.3 Correctness of final transformations

Correctness of both `norm-t` and `norm-l` is rather obvious due to the simple nature of these transformations. But we still state the correctness theorems to highlight the side conditions and progress claims (the details of the proofs could be found in the code).

The progress lemma for `norm-l` states that after the transformation there are no rules with right-hand sides of more than two symbols.

```
nl-progress :  (Rs : Rules) → (A : N)
    → (rhs : RHS) →  A ⟶ rhs ∈ norm-l Rs
    → length rhs ≤ 2
```

The progress lemma for `norm-t` states that, for any `Rs` for all rules `A ⟶ rhs ∈ norm-t Rs`, either `rhs ≡ [ tm a ]` for some terminal `a` or `rhs` consists of nonterminals only.

```
nt-progress : (Rs : Rules) → (A : N)
    → (rhs : RHS) → A ⟶ rhs ∈ norm-t Rs
    → ∃(a : T) rhs ≡ [ tm a ] ∨ ntOnly rhs
```

Next, `nl-snd` and `nt-snd` state that each tree for normalized grammar that is rooted by some nonterminal present in the original grammar can be transformed into a tree for the original grammar:

```
nl-snd : (Rs : Rules) → (A : N)
   → (s : String) → A ∈ NTs Rs
   → Tree (norm-l Rs) A s → Tree Rs A s

nt-snd : (Rs : Rules) → (A : N)
  → (s : String) → A ∈ NTs Rs
  → Tree (norm-t Rs) A s → Tree Rs A s
```

The side condition `A ∈ NTs Rs` is important, because a tree rooted by some "freshly" added nonterminal has no corresponding tree in the original grammar, where the fresh nonterminal is not present.

Conversely, any parse tree for `Rs` could be mapped to parse trees for `norm-l Rs` and `norm-t Rs`.

```
nl-cmplt : (Rs : Rules) → (A : N) → (s : String)
    → Tree Rs A s → Tree (norm-l Rs) A s

nt-cmplt : (Rs : Rules) → (A : N) → (s : String)
    → Tree Rs A s → Tree (norm-t Rs) A s
```

6. Full normalization and correctness

6.1 Full normalization function

Finally, we are ready to define the full normalization function:

```
norm : Rules → Rules
norm = norm-u ∘ norm-e ∘ norm-t ∘ norm-l
```

The function `norm` is a composition of the four transformations we have introduced. The order in which these transformations are chained matters. For example, `norm-e` can add new unit rules, so `norm-u` must be performed after `norm-e`.

Progress The question of progress of `norm` boils down to the questions about preservation of the progress properties of individual constituent transformations by those transformations that follow:

1. Since `norm-t` never increases the length of the right-hand side of any rule, `norm-t` preserves the progress made by `norm-l`. We prove that, if the right hand side of every rule in `Rs` is shorter that some `n : ℕ`, then the same holds for all rules in `norm-t Rs`:

   ```
   nt-efct : (Rs : Rules) → (n : ℕ)
        → ((A : N) → (rhs : RHS)
                   → A ⟶ rhs ∈ Rs
                   → length rhs ≤ n)
        → (A : N) → (rhs : RHS)
        → A ⟶ rhs ∈ norm-t Rs
        → length rhs ≤ n
   ```

2. We show that, if `A ⟶ rhs ∈ norm-e Rs`, then `rhs` must be a subsequence of some `rhs'` such that `A ⟶ rhs' ∈ Rs`. Since the progress properties of `norm-l` and `norm-t` are closed under the subsequence relation, `norm-e` preserves the progress achieved by `norm-l` and `norm-t`:

   ```
   ne-efct : (Rs : Rules) → (A : N) → (rhs : RHS)
     → A ⟶ rhs ∈ norm-e Rs → A ⟶ rhs ∉ Rs
     → ∃(rhs' : RHS) A ⟶ rhs' ∈ Rs ×
         rhs ∈ allSubSeq (∈ nullables Rs) rhs'
   ```

3. Since `norm-u` does not introduce any new right-hand sides into a grammar, it preserves the progress properties of all other transformations. Formally, we prove that, if there is some predicate that holds for all RHSs in the grammar `Rs`, then it will also hold for all RHSs in the grammar `norm-u`:

   ```
   nu-efct : (P : RHS → Set) → (Rs : Rules)
     → ((A : N) → (rhs : RHS)
                → A ⟶ rhs ∈ Rs → P rhs)
     → (A : N) → (rhs : RHS)
     → A ⟶ rhs ∈ norm-u Rs
     → P rhs
   ```

Finally, we show the following progress property of `norm`:

```
norm-progress : (Rs : Rules) → (A : N)
    → (rhs : RHS) → A ⟶ rhs ∈ norm Rs
    → (∃(B C : N) rhs ≡ [ nt B, nt C ]) ∨
       (∃(a : T) rhs ≡ [ tm a ])
```

It states that, for any rule `A ⟶ rhs ∈ norm Rs`, either `rhs ≡ [ nt B, nt C ]` for some nonterminals `B` and `C` or `rhs ≡ [ tm a ]` for some terminal `a`.

Soundness To show soundness of `norm`, we only need to chain the soundness results of the individual transformations:

```
norm-snd : (Rs : Rules) → (A : N)
  → (s : String)  → A ∈ NTs Rs
  → Tree (norm Rs) A s → Tree Rs A s
```

Completeness As in the case of soundness, completeness of `norm` is proved by chaining the completeness results of the small transformations:

```
norm-cmplt : (Rs : Rules) → (A : N)
  → (s : String) → s ≢ []
  → Tree Rs A s  → Tree (norm Rs) A s
```

6.2 Grammars with a start nonterminal

Now, we define a context-free grammar as a set of rules with a fixed start nonterminal:

```
record Grammar : Set where
  field
    S : N
    Rs : Rules
```

(Given some `G : Grammar`, we write `S G` and `Rs G` for projections of the start terminal and the list of rules respectively.)

Next, the language of the grammar `G` is defined as:

```
TreeS : Grammar → String → Set
TreeS G s = Tree (Rs G) (S G) s
```

Next, we implement normalization of context-free grammars:

```
normS : Grammar → Grammar
normS G = record {
  S  = S';
  Rs = if S G ∈ nullables (Rs G)
       then S' ⟶ [] :: Rs'
       else Rs'
  }
   where
     S'  = newnt (Rs G)
     Rs' = norm ((S' ⟶ [ nt (S G) ]) :: Rs G)
```

To normalize a context-free grammar we have the following algorithm:

1. Declare `newnt (Rs G)` as a new starting nonterminal.
2. Normalize the set of rules `Rs G` extended by the rule `newnt (Rs G) ⟶ [ nt (S G) ]`. Since `newnt (Rs G)` is fresh, it is clear that its language is same as the language of nonterminal `S G` and it will not affect the language of any other nonterminal (this step guarantees that new starting nonterminal does not appear on the right hand sides of the rules).
3. Finally, if the starting nonterminal of the original grammar was nullable then add the rule `newnt (Rs G) ⟶ []` to the normalized set of rules to retain the empty string in the language of normalized grammar. Intuitively, it is safe to do so, because `new (Rs G)` does not appear in the right-hand sides of the other rules.

Let us look at the final versions of progress, soundness and completeness properties:

Progress

```
normS-progress : (G : Grammar) → (A : N)
  → (rhs : RHS)
  → let G' = normS G in A ⟶ rhs ∈ Rs G'
  → (∃(B C : N) rhs ≡ [ nt B, nt C ]
        × B ≢ S G'
        × C ≢ S G') ∨
     (∃(a : T) rhs ≡ [ tm a ])  ∨
     (rhs ≡ [] × A ≡ S G')
```

For any rule `A ⟶ rhs ∈ Rs (normS G)`, the right-hand side `rhs` is either `[ nt B, nt C ]` for some nonterminals `B` and `C` where neither `B` nor `C` are starting nonterminals or `[ tm a ]` for some terminal `a`, or `[]` with the condition that `A ≡ S (normS G)`.

Soundness and completeness

```
normS-snd : (G : Grammar) → (s : String)
   → S G ∈ NTs (Rs G)
   → TreeS (normS G) s → TreeS G s

normS-cmplt : (G : Grammar) → (s : String)
   → TreeS G s → TreeS (normS G) s
```

If a given grammar is well-formed (i.e., the start nonterminal actually appears in the given list of rules), then normalization preserves the language of the grammar.

7. Related Work and Conclusions

While a number of authors have formalized various parts of the theory of regular grammars or expressions and finite automata, efforts in the direction of context-free grammars seem fewer.

Several authors have considered parsing of context-free grammars. Barthwal and Norrish [6] formalized SLR parsing with the HOL4 theorem prover. Ridge [10] has formalized the correctness of a general CFG parser constructor in HOL4.

Koprowski and Binsztok [4] have formalized parsing expression grammars (PEGs), a formalism for specifying recursive descent parses, in Coq. Jourdan, Pottier and Leroy [7] have presented a validator that checks if a context-free grammar and an LR(1) parser agree; they have proved the validator correct in Coq.

Danielsson [2] has implemented a library of parser combinators in Agda treating left recursion with coinduction. Sjöblom [11] has formalized an aspect of Valiant's parsing algorithm.

Regarding normalization of context-free grammars, Barthwal and Norrish [5] described a formalisation of the Chomsky and Greibach normal forms for context-free grammars with the HOL4 theorem prover. They showed how to solve the problems which arise from mechanising the straightforward pen and paper proofs. The non-constructive setting gave the advantage of the power of extensional and classical reasoning, but also the significant drawback that it did not deliver actual functions for normalizing grammars or converting parse trees between grammars.

We have proved in Agda that a general CFG and its Chomsky normal form accept the same language. As a program, the proof consists of functions for conversion of parse trees between the original and normalized grammars. This is a typical added benefit of formalization in a language like Agda; e.g., a proof that a CFG and the corresponding pushdown automaton accept the same language would give functions for conversion between parse trees and accepting runs.

Combined with the CYK parser we have written previously [3], the code of this paper gives us a parser for CFGs in general form. There is, however, a caveat: we do not get all parse trees of the grammar; moreover, it is not entirely obvious which parse trees we get and which are lost.

To make this precise, we plan to extend this work as follows. Instead of unnamed rules (a rule is identified by the left-hand nonterminal and the right-hand list of symbols), we name rules. This gives us finer control over parse trees. Now we expect that the conversion of a parse tree in the normalized grammar to the original grammar and back again will be identity while the conversion of a parse of the original grammar to the normalized grammar and back will be an idempotent function—a kind of normalizer of parse trees that truncates nullable paths and removes unit cycles. Normalization of parse trees by passing through the normalized grammar can then be seen as a form of normalization-by-evaluation.

Overall, the constructive approach allows one to give parse trees a first-class status: knowing that a string is in a language includes knowing a proof of this, i.e., a parse tree. These proofs become objects of analysis and manipulation.

Acknowledgement We thank our anonymous referees for the useful feedback. This research was supported by the ERDF funded Estonian CoE project EXCS and the Estonian Research council target-financed research theme No. 0140007s12 and grant No. 9475.

References

[1] J. E. Hopcroft, J. D. Ullman. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979.

[2] N. A. Danielsson. Total parser combinators. In *Proc. of 15th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '10*, pp. 285–296. ACM, 2010.

[3] D. Firsov, T. Uustalu. Certified CYK parsing of context-free languages. *J. of Log. and Algebr. Meth. in Program.*, v. 83(5–6), pp. 459–468, 2014.

[4] A. Koprowski, H. Binsztok. TRX: A formally verified parser interpreter. *Log. Meth. in Comput. Sci.*, v. 7(2), article 18, 2011.

[5] A. Barthwal, M. Norrish. A formalisation of the normal forms of context-free grammars in HOL4. In A. Dawar, H. Veith, eds., *Proc. of 24th Int. Wksh. on Computer Science Logic, CSL 2010*, v. 6247 of *Lect. Notes in Comput. Sci.*, pp. 95–109. Springer, 2010.

[6] A. Barthwal, M. Norrish. Verified, executable parsing. In G. Castagna, ed., *Proc. of 18th Europ. Symp. on Programming, ESOP 2009*, v. 5502 of *Lect. Notes in Comput. Sci.*, pp. 160–174. Springer, 2009.

[7] J.-H. Jourdan, F. Pottier, X. Leroy. Validating LR(1) parsers. In H. Seidl, ed., *Proc. of 21st Europ. Symp. on Programming, ESOP 2012*, v. 7211 of *Lect. Notes in Comput. Sci.*, pp. 397–416. Springer, 2012.

[8] Y. Minamide. Verified decision procedures on context-free grammars. In K. Schneider, J. Brandt, eds., *Proc. of 20th Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLS 2007*, v. 4732 of *Lect. Notes in Comput. Sci.*, pp. 173–188. Springer, 2007.

[9] U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, S. D. Swierstra, eds., *Revised Lectures from 6th Int. School on Advanced Functional Programming, AFP 2008*, v. 5832 of *Lect. Notes in Comput. Sci.*, pp. 230–266. Springer, 2009.

[10] T. Ridge. Simple, functional, sound and complete parsing for all context-free grammars. In J.-P. Jouannaud, Z. Shao, eds., *Proc. of 1st Int. Conf. on Certified Programs and Proofs, CPP 2011*, v. 7086 of *Lect. Notes in Comput. Sci.*, pp. 103–118. Springer, 2011.

[11] T. B. Sjöblom. An Agda proof of the correctness of Valiant's algorithm for context free parsing. Master's thesis, Dept. of Computer Sci. and Engin., Chalmers University of Technology, 2013.

Curriculum Vitae

1. Personal data

Name	Denis Firsov
Date and place of birth	28 May 1987, Mustvee
Citizenship	Estonian
E-mail address	denis@cs.ioc.ee

2. Education

Educational institution	Period	Degree
Tallinn University of Technology	2012–2016	PhD studies
Tallinn University of Technology	2010–2012	Master of Science
Estonian Information Technology College	2006–2010	Diploma

3. Language skills

Russian	native
English	fluent
Estonian	fluent

4. Special courses

Period	Event
27 June–1 July 2016	Second International Summer School on Behavioural Types
28 Feb.–4 Mar. 2016	21st Estonian Winter School in Computer Science
13–22 July 2015	Summer School on Complexity and Concurrency through Topology
6–10 July 2015	Oxford Summer School on Generic and Effectful Programming
1–6 Mar. 2015	20th Estonian Winter School in Computer Science
20–27 Apr. 2014	Midlands Graduate School 2014
2–7 Mar. 2014	19th Estonian Winter School in Computer Science
8–20 July 2013	Domain Specific Languages Summer School 2013
8–12 Apr. 2013	Midlands Graduate School 2013
3–8 Mar. 2013	18th Estonian Winter School in Computer Science
16–28 July 2012	Oregon Programming Languages Summer School
19–23 Aug. 2012	11th Estonian Summer School in Computer and Systems Science

5. Professional employment

Period	Organisation	Position
2011– . . .	Inst. of Cybernetics at TUT	engineer, junior researcher
2010–2011	Attitude OÜ	software architect
2009–2010	Majandustarkvara OÜ	software developer

6. Research activity

   Algorithms, functional programming languages, type systems, constructive mathematics, software verification.

7. Publications

   1. D. Firsov, T. Uustalu. Certified parsing of regular languages. In G. Gonthier, M. Norrish, eds., *Proc. of 3rd Int. Conf. on Certified Programs and Proofs, CPP 2013 (Melbourne, Dec. 2013)*, v. 8307 of *Lect. Notes in Comput. Sci.*, pp. 98–113. Springer, 2013.
   2. D. Firsov, T. Uustalu. Certified CYK parsing of context-free languages. *J. of Log. and Algebr. Meth. in Program.*, v. 83(5–6), pp. 459–468, 2014.
   3. D. Firsov, T. Uustalu. Certified normalization of context-free grammars. In *Proc. of 4th ACM SIGPLAN Conf. on Certified Programs and Proofs, CPP '15 (Mumbai, Jan. 2015)*, pp. 167–174. ACM Press, 2015.
   4. D. Firsov, T. Uustalu. Dependently typed programming with finite sets. In *Proc. of 2015 ACM SIGPLAN Wksh. on Generic Programming, WGP '15 (Vancouver, BC, Aug. 2015)*, pp. 33–44. ACM Press, 2015.
   5. D. Firsov, T. Uustalu. Acyclic attribute evaluation in a dependently typed setting. In *Abstracts of 27th Nordic Workshop on Programming Theory, NWPT 2015*, Technical report RUTR-SCS16001, School of Computer Science, pp. 124–126, Reykjavik University, 2016.
   6. D. Firsov, T. Uustalu, N. Veltri. Variations on Noetherianness. In R. Atkey, N. Krishnaswamy, eds., *Proc. of 6th Wksh. on Mathematically Structured Functional Programming, MSFP 2016 (Eindhoven, 2016)*, *Electron. Proc. in Theor. Comput. Sci.*, pp. 76–88. Open Publishing Assoc., 2016.
   7. D. Firsov, W. Jeltsch. Purely functional incremental computing. In F. Castor, Y. D. Liu, eds., *Proc. of 20th Brazilian Symp. on Programming Languages, SBLP 2016 (Maringá, Paraná, Sept. 2016)*, *Lect. Notes in Comput. Sci.*, Springer, to appear.

Elulookirjeldus

1. Isikuandmed

Ees- ja perekonnanimi	Denis Firsov
Sünniaeg ja -koht	28.05.1987, Mustvee
Kodakondsus	Eesti
E-posti aadress	denis@cs.ioc.ee

2. Hariduskäik

Õppeasutus	Lõpetamisaeg	Haridus
Tallinna Tehnikaülikool	2012–2016	Doktoriõpe
Tallinna Tehnikaülikool	2010–2012	Magistrikraad
Eesti Infotehnoloogia Kolledž	2006–2010	Rakenduskõrgharidus

3. Keelteoskus

Vene keel	emakeel
Inglise keel	kõrgtase
Eesti keel	kõrgtase

4. Täiendusõpe

Period	Event
27.06–1.07.16	Second International Summer School on Behavioural Types
28.02–4.03.16	21st Estonian Winter School in Computer Science
13–22.07.15	Summer School on Complexity and Concurrency through Topology
6–10.07.15	Oxford Summer School on Generic and Effectful Programming
1–6.03.15	20th Estonian Winter School in Computer Science
20–27.04.14	Midlands Graduate School 2014
2–07.03.14	19th Estonian Winter School in Computer Science
8–20.07.13	Domain Specific Languages Summer School 2013
8–12.04.13	Midlands Graduate School 2013
3–8.03.13	18th Estonian Winter School in Computer Science
16–28.07.12	Oregon Programming Languages Summer School
19–23.08.12	11th Estonian Summer School on Computer and Systems Science

5. Teenistuskäik

Töötamise aeg	Tööandja nimetus	Ametikoht
2011– . . .	TTÜ Küberneetika Instituut	insener, nooremteadur
2010–2011	Attitude OÜ	tarkvaraarhitekt
2009–2010	Majandustarkvara OÜ	tarkvaraarendaja

6. Teadustegevus

   Algoritmid, funktsionaalsed programmeerimiskeeled, tüübisüsteemid, konstruktiivne matemaatika, tarkvara verifitseerimine.

7. Publikatsioonid

   1. D. Firsov, T. Uustalu. Certified parsing of regular languages. In G. Gonthier, M. Norrish, eds., *Proc. of 3rd Int. Conf. on Certified Programs and Proofs, CPP 2013 (Melbourne, Dec. 2013)*, v. 8307 of *Lect. Notes in Comput. Sci.*, pp. 98–113. Springer, 2013.
   2. D. Firsov, T. Uustalu. Certified CYK parsing of context-free languages. *J. of Log. and Algebr. Meth. in Program.*, v. 83(5–6), pp. 459–468, 2014.
   3. D. Firsov, T. Uustalu. Certified normalization of context-free grammars. In *Proc. of 4th ACM SIGPLAN Conf. on Certified Programs and Proofs, CPP '15 (Mumbai, Jan. 2015)*, pp. 167–174. ACM Press, 2015.
   4. D. Firsov, T. Uustalu. Dependently typed programming with finite sets. In *Proc. of 2015 ACM SIGPLAN Wksh. on Generic Programming, WGP '15 (Vancouver, BC, Aug. 2015)*, pp. 33–44. ACM Press, 2015.
   5. D. Firsov, T. Uustalu. Acyclic attribute evaluation in a dependently typed setting. In *Abstracts of 27th Nordic Workshop on Programming Theory, NWPT 2015*, Technical report RUTR-SCS16001, School of Computer Science, pp. 124–126, Reykjavik University, 2016.
   6. D. Firsov, T. Uustalu, N. Veltri. Variations on Noetherianness. In R. Atkey, N. Krishnaswamy, eds., *Proc. of 6th Wksh. on Mathematically Structured Functional Programming, MSFP 2016 (Eindhoven, 2016)*, *Electron. Proc. in Theor. Comput. Sci.*, pp. 76–88. Open Publishing Assoc., 2016.
   7. D. Firsov, W. Jeltsch. Purely functional incremental computing. In F. Castor, Y. D. Liu, eds., *Proc. of 20th Brazilian Symp. on Programming Languages, SBLP 2016 (Maringá, Paraná, Sept. 2016)*, *Lect. Notes in Comput. Sci.*, Springer, to appear.

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo**. Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pavel Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. **Gunnar Piho**. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. **Juri Gavšin**. Intrinsic Robot Safety Through Reversibility of Actions. 2011.

69. **Dmitri Mihhailov**. Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models. 2012.

70. **Anton Tšertov**. System Modeling for Processor-Centric Test Automation. 2012.

71. **Sergei Kostin**. Self-Diagnosis in Digital Systems. 2012.

72. **Mihkel Tagel**. System-Level Design of Timing-Sensitive Network-on-Chip Based Dependable Systems. 2012.

73. **Juri Belikov**. Polynomial Methods for Nonlinear Control Systems. 2012.

74. **Kristina Vassiljeva**. Restricted Connectivity Neural Networks based Identification for Control. 2012.

75. **Tarmo Robal**. Towards Adaptive Web – Analysing and Recommending Web Users` Behaviour. 2012.

76. **Anton Karputkin**. Formal Verification and Error Correction on High-Level Decision Diagrams. 2012.

77. **Vadim Kimlaychuk**. Simulations in Multi-Agent Communication System. 2012.

78. **Taavi Viilukas**. Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits. 2012.

79. **Marko Kääramees**. A Symbolic Approach to Model-based Online Testing. 2012.

80. **Enar Reilent**. Whiteboard Architecture for the Multi-agent Sensor Systems. 2012.

81. **Jaan Ojarand**. Wideband Excitation Signals for Fast Impedance Spectroscopy of Biological Objects. 2012.

82. **Igor Aleksejev**. FPGA-based Embedded Virtual Instrumentation. 2013.

83. **Juri Mihhailov**. Accurate Flexible Current Measurement Method and its Realization in Power and Battery Management Integrated Circuits for Portable Applications. 2013.

84. **Tõnis Saar**. The Piezo-Electric Impedance Spectroscopy: Solutions and Applications. 2013.

85. **Ermo Täks**. An Automated Legal Content Capture and Visualisation Method. 2013.

86. **Uljana Reinsalu**. Fault Simulation and Code Coverage Analysis of RTL Designs Using High-Level Decision Diagrams. 2013.

87. **Anton Tšepurov**. Hardware Modeling for Design Verification and Debug. 2013.

88. **Ivo Müürsepp**. Robust Detectors for Cognitive Radio. 2013.

89. **Jaas Ježov**. Pressure sensitive lateral line for underwater robot. 2013.

90. **Vadim Kaparin**. Transformation of Nonlinear State Equations into Observer Form. 2013.

92. **Reeno Reeder**. Development and Optimisation of Modelling Methods and Algorithms for Terahertz Range Radiation Sources Based on Quantum Well Heterostructures. 2014.

93. **Ants Koel**. GaAs and SiC Semiconductor Materials Based Power Structures: Static and Dynamic Behavior Analysis. 2014.

94. **Jaan Übi**. Methods for Coopetition and Retention Analysis: An Application to University Management. 2014.

95. **Innokenti Sobolev**. Hyperspectral Data Processing and Interpretation in Remote Sensing Based on Laser-Induced Fluorescence Method. 2014.

96. **Jana Toompuu**. Investigation of the Specific Deep Levels in *p*-, *i*- and *n*-Regions of GaAs p^+-*pin*-n^+ Structures. 2014.

97. **Taavi Salumäe**. Flow-Sensitive Robotic Fish: From Concept to Experiments. 2015.

98. **Yar Muhammad**. A Parametric Framework for Modelling of Bioelectrical Signals. 2015.

99. **Ago Mõlder**. Image Processing Solutions for Precise Road Profile Measurement Systems. 2015.

100. **Kairit Sirts**. Non-Parametric Bayesian Models for Computational Morphology. 2015.

101. **Alina Gavrijaševa**. Coin Validation by Electromagnetic, Acoustic and Visual Features. 2015.

102. **Emiliano Pastorelli**. Analysis and 3D Visualisation of Microstructured Materials on Custom-Built Virtual Reality Environment. 2015.

103. **Asko Ristolainen**. Phantom Organs and their Applications in Robotic Surgery and Radiology Training. 2015.

104. **Aleksei Tepljakov**. Fractional-order Modeling and Control of Dynamic Systems. 2015.

105. **Ahti Lohk**. A System of Test Patterns to Check and Validate the Semantic Hierarchies of Wordnet-type Dictionaries. 2015.

106. **Hanno Hantson**. Mutation-Based Verification and Error Correction in High-Level Designs. 2015.

107. **Lin Li**. Statistical Methods for Ultrasound Image Segmentation. 2015.

108. **Aleksandr Lenin**. Reliable and Efficient Determination of the Likelihood of Rational Attacks. 2015.

109. **Maksim Gorev**. At-Speed Testing and Test Quality Evaluation for High-Performance Pipelined Systems. 2016.

110. **Mari-Anne Meister**. Electromagnetic Environment and Propagation Factors of Short-Wave Range in Estonia. 2016.

111. **Syed Saif Abrar**. Comprehensive Abstraction of VHDL RTL Cores to ESL SystemC. 2016.

112. **Arvo Kaldmäe**. Advanced Design of Nonlinear Discrete-time and Delayed Systems. 2016.

113. **Mairo Leier**. Scalable Open Platform for Reliable Medical Sensorics. 2016.

114. **Georgios Giannoukos**. Mathematical and Physical Modelling of Dynamic Electrical Impedance. 2016.

115. **Aivo Anier**. Model Based Framework for Distributed Control and Testing of Cyber-Physical Systems. 2016.