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 $\texttt{S-axiom}_1$ or $\texttt{S-axiom}_2$ 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 $i \leq n \leq i + j$ 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 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.