Certification of context-free grammar algorithms

Denis Firsov

Institute of Cybernetics at TUT

August 31, 2016

Certification

Certification refers to the confirmation of certain characteristics of an object, person, or organization. The confirmation is often provided by some form of review, assessment, or audit.

(Wikipedia)

Software certification

The correctness of a program is established by full formal verification:

- The specification of the program is presented in some rigorous mathematical language.
- The program itself also must be modeled in some mathematical formalism.
- The verification is done by providing a formal proof that the model satisfies the specification.
- The validity of the formal proof is checked by a computer program.

Correct algorithm $\neq$ correct implementation

- Binary search algorithm was first described in 1946, but the first implementation of binary search without bugs was published in 1962 (TAOCP, Volume 3, Section 6.2.1).
- In 2015, de Gouw et al. investigated the correctness of Java sorting. The result was a proof that *java.utils.Collection.sort()* is broken (by an explicit example) and a proposal for fixing it.

Examples

- The CompCert project (Leroy et al., 2006) performed formal verification of a C compiler in Coq (5 years; 42k lines of Coq).
- The seL4 project (Klein et al., 2010) certified an OS kernel. The project dealt with 10k lines of C code and 200k lines of proofs in Isabelle/HOL showing safety against code injection, buffer overflow, general exceptions, memory leaks, etc.
- The original proof of the "Four color theorem" was partly generated by a program (written in a general purpose language) and was not generally accepted by mathematicians as "infeasible for a human to check by hand". In 2005, Benjamin Werner and Georges Gonthier formalized the proof in the Coq proof assistant.

Dependently typed programming

- The Curry–Howard correspondence is the central observation that proof systems and models of computation are structurally the same kind of object.
- The main idea: a proof is a functional program, the formula it proves is the type of the program.
- In this work, we use the dependently typed functional programming language Agda. It acts both as a proof framework and as a functional programming language with an expressive type system.
- Examples:

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

```
lemma : (m n : ℕ) → m > n → ∃[ k : ℕ ] k + n ≡ m
lemma = ...
```

Compilation

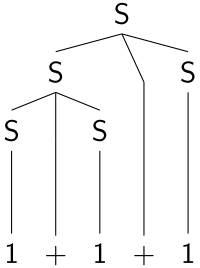
- Compilation, as a process of translating a program written in a high-level language into a machine language, consists of a number of phases:
  1. lexical analysis,
  2. syntax analysis,
  3. semantic analysis,
  4. optimisation,
  5. code generation.
- Crucially, low-level code produced as output must have the same semantics as the high-level code taken as input.
- The CompCert project certified a C compiler starting from semantic analysis (handful of bugs were found later in unverified parts).
- In my master thesis, I implemented a certified parser for regular language (lexical analysis).

Syntax analysis

- A context-free grammar is a 4-tuple $G = (N, T, R, S)$:
  - N is a finite set of nonterminals.
  - T is a finite set of terminals.
  - R is a finite set of production rules. A rule is usually denoted by an arrow as $A \longrightarrow \gamma$, where $A \in N$ and γ is a sequence of nonterminals and terminals.
  - S is the start nonterminal from the set N.
- Let $\alpha A \beta$ be some sequence of symbols, and A be a nonterminal. If there is a rule $A \longrightarrow \gamma$ in R then we can *derive* $\alpha\gamma\beta$ from $\alpha A \beta$.
- Then the *language* of the grammar G is the set of all strings (sequences of terminals) derivable from the nonterminal S.
- In Agda notation we have:
  - (Global) types `N`, `T`, and finite `R`.
  - A grammar type `Grammar`. (The start nonterminal is not necessarily fixed.)
  - A parse tree type `Tree G A s`

Syntax analysis – example

- Consider the grammar G, with $N = \{S\}$, $T = \{1, +\}$, and $R = \{S \longrightarrow 1, S \longrightarrow S + S\}$.
- Then the following is a possible derivation tree of the string "1+1+1":




```
: Tree G S "1+1+1"
```

Problem statement

The main interest is to implement a certified function that, given a context-free grammar and a string, finds a derivation (parse tree) of the string in the grammar provided.

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.

Finite sets constructively

- In constructive logic there are many different definitions of finite sets which collapse classically (Kuratowski finite, Dedekind finite, Noetherian sets, Streamless sets, etc.).
- From the programming standpoint, the important notion of finiteness is listability of a set:

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

Properties of listability

- An important observation is that listable sets have decidable equality:

```
lstbl2eq : {X : Set} → Listable X
  → (x₁ x₂ : X) → x₁ ≡ x₂ ⊎ ¬ x₁ ≡ x₂
```

- For any set `X` there is a surjection from an initial segment of natural numbers to `X` if and only if `X` is listable.
- For any set `X` there is a bijection from an initial segment of natural numbers to `X` if and only if `X` is listable.

Pragmatic finite subsets

- We define a new type `FinSubDesc` which is parameterized by some base set `U`, a decidable equality on its elements, and a Boolean flag.

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

- A subset is described by listing its elements, e.g.:

```
ℕ-subset : FinSubDesc ℕ _≟_ true
ℕ-subset = fsd-plain (1 ∷ 2 ∷ 3 ∷ [])
```

- Such a description defines a subset of `U`:

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

Properties of pragmatic finite subsets

- The subset defined is listable: We have a list of elements . . .

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

- . . . and it is complete:

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

Listable subsets, predicate matching, and prover

- We also formalized the notion of listable subset.
- We proved that listable subsets generalize listable sets.
- We showed that listable subsets do not imply decidable equality.

- We described the necessary and sufficient conditions to treat lists of type `List ((X → Bool) × (X → Y))` as functions on `X` defined in a piecewise manner.

- We designed combinators that decide existential and universal statements over decidable properties on finite sets.

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.

Chomsky normal form

- 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 the start nonterminal. There must be a flag (`nullable`) which indicates if the empty word is in the language of G.
- Every string has a finite number of parse trees for a CNF grammar.
- Parsing is conceptually simple with CNF grammars.

<table>
<tr><td colspan="6">A</td></tr>
<tr><td colspan="3">B</td><td colspan="3">C</td></tr>
<tr><td colspan="3">...</td><td colspan="3">...</td></tr>
<tr><td>s_0</td><td>...</td><td>s_{k-1}</td><td>s_k</td><td>...</td><td>s_n</td></tr>
</table>

- In this paper we work with one fixed grammar `G` and some predicate `isCNF` which holds for `G`.

CYK parsing

s_0	...	...	...	s_n	
...	...	...	...	...	$P_{0,n+1}$
	...	...	...	...	...
		$P_{i,i}$	$P_{i,i+1}$	...	...
			...	...	...
				...	...
					...

- A linear representation of a matrix:

$$\texttt{Mtrx s = List (}\exists\texttt{[i : }\mathbb{N}\texttt{] }\exists\texttt{[j : }\mathbb{N}\texttt{]}$$
$$\exists\texttt{[A : N] Tree G A s[i, j])}$$

- We can combine two matrices

$$\texttt{m}_1 \texttt{ * m}_2 \texttt{ = } \{ \texttt{ (i, j, A, cons t}_1\texttt{ t}_2\texttt{) | (i, k, B, t}_1\texttt{) } \leftarrow \texttt{ m}_1\texttt{,}$$
$$\texttt{(k, j, C, t}_2\texttt{) } \leftarrow \texttt{ m}_2\texttt{, (A } \longrightarrow \texttt{ BC) } \leftarrow \texttt{ R } \}$$

Certified CYK parsing

- Computing parse trees for substrings of a particular length:

```
pow : (s : String) → ℕ → Mtrx s
pow s 0 = { (i, i, S, empt i | nullable, i ← [1 ... n) }
pow s 1 = { (i, 1+i, A, sngl p) | p : A ⟶ sᵢ ∈ R }
pow s n = { t | k ← [1 ... n),
                t ← pow s k * pow s (n - k) }
```

- We prove that pow is complete:

```
pow-complete : (s : String) → (X : N)
  → (t : Tree G X s)
  → (0, length s, X, t) ∈ pow s (length s)
```

- The string is in the language if there is a parse tree from starting nonterminal.

```
cyk-parse : (s : String) → List (Tree G S s)
cyk-parse s = { t | (_, _, S, t) ← pow s (length s) }
```

Certified CYK parsing – termination

- We formalize the idea of well-founded relations by using the concept of accessibility:

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

- A relation is *well-founded*, if all carrier set elements are accessible.

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

- We prove that the $<$ relation on natural numbers is well-founded.

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

- The recursive calls of the `pow` function are made along this well-founded relation.

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

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

Certified CYK parsing – memoization

- We define certified memoization tables:

```
MemTbl s = (n : ℕ) → ∃[m' : Mtrx s] m' ≡ pow n
```

- The function `pow-tbl` uses the table in place of recursive calls:

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

- We can update the table at a particular position:

```
updateTbl : {s : String} → MemTbl s → ℕ → MemTbl s
updateTbl tbl' e l = if l ≠ e then tbl' l
  else (pow-tbl l tbl', pow≡pow-tbl)
```

- Finally, we gradually fill the table:

```
pow-mem : {s : String} → ℕ → MemTbl s → Mtrx s
pow-mem n tbl = foldl updateTbl tbl [2..n] $ n
```

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.

Normalization of CFGs

Every context-free grammar can be transformed into an equivalent one in Chomsky normal form. This is accomplished by a sequence of four transformations.

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 $\mathtt{A} \longrightarrow \mathtt{X}_1\mathtt{A}_1$, $\mathtt{A}_1 \longrightarrow \mathtt{X}_2\mathtt{A}_2$, $\mathtt{A}_{k-2} \longrightarrow \mathtt{X}_{k-1}\mathtt{X}_k$ where $\mathtt{A}_i$ are “fresh” nonterminals;
4. for each terminal a, adding a new rule $\mathtt{A} \longrightarrow$ a where $\mathtt{A}$ is a fresh nonterminal and replacing a in the right-hand sides of all rules with length at least two with $\mathtt{A}$.

Elimination of unit rules

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

```
nu-step : Grammar → N → Grammar
nu-step G A = { A ⟶ rhs |
                A ⟶ rhs ∈ G ∪ aux,  rhs ≠ A }
  where
    aux = { X ⟶ rhs | X ⟶ A ∈ G,
                      A ⟶ rhs ∈ G }
```

- 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)
```

Correctness of elimination of unit rules

- First, we showed that `nu-step` achieves some progress towards normality of the grammar:

```
step-progress : (G : Grammar) → (A B : N)
   → (A ⟶ B) ∉ nu-step G B
```

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

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

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

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

- There is a straightforward lifting of this lemmas to `norm-u`.

Overall normalization and correctness

- The full normalization function is defined by composition:

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

- We proved soundness, completeness, and progress of all constituent transformations.
- Additionally, we showed that later stages preserve the progress of earlier transformations.

Correctness of full normalization

- The `norm` functions achieves Chomsky normal form:

```
norm-progress : (G : Grammar) → isCNF (norm G)
```

- The `norm` function is sound and complete (`S` and `S'` are start nonterminals of `G` and `norm G`):

```
sound : (G : Grammar) → Tree (norm G) S' s → Tree G S s

compl : (G : Grammar) → Tree G S s → Tree (norm G) S' s
```

General context-free parsing

- The CYK algorithm, normalization function, and the proof of soundness can be combined to give a general context-free parsing function:

```
parse : (G : Grammar) → (s : String) → List (Tree G S s)
parse G s = map sound cykL
 where
   cnfG = norm G
   cykL = cyk-parse cnfG (norm-progress G) s
```

- Finally, the completeness of CYK implementation together with completeness of normalization induce the completeness of a `parse`:

```
parse-complete : (G : Grammar) → (s : String)
  → Tree G S s → ∃[ t’ : Tree G S s ] t’ ∈ parse G s
```

Conclusions

- Programming with dependent types allows one to design datastructures and functions which are correct-by-construction.
- In this thesis, we demonstrated this by formalizing the theory of context-free languages.
- We studied listability of sets in Agda and implemented viable solutions to boilerplate-free programming with listable sets.
- We used refinement techniques to implement the certified CYK parsing algorithm for context-free grammars in Chomsky normal form.
- We implemented a certified normalization procedure for context-free grammars.
- 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.
- The 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.

Beware of bugs in the above code; I have only proved it correct, not tried it.

(Donald Knuth)