Certified Grammar Transformation to Chomsky Normal Form in F*

M.I. Polubelova <polubelovam@gmail.com>
S.N. Bozhko <gkerfimf@gmail.com>
S.V. Grigorev <Semen.Grigorev@jetbrains.com>
Saint Petersburg State University,
7/9, Universitetskaya Nab., St. Petersburg, 199034, Russia

Abstract. Certified programming allows to prove that the program meets its specification. The check of correctness of a program is performed at compile time, which guarantees that the program always runs as specified. Hence, there is no need to test certified programs to ensure they work correctly. There are numerous toolchains designed for certified programming, but F* is the only language that support both general-purpose programming and semi-automated proving. The latter means that F* infers proofs when it is possible and a user can specify more complex proofs if necessary. We work on the application of this technique to a grammarware research and development project YaccConstructor. We present a work in progress verified implementation of transformation of Context-free grammar to Chomsky normal form, that is making progress toward the certification of the entire project. Among other features, F* system allows to extract code in F# or OCaml languages from a program written in F*. YaccConstructor project is mostly written in F#, so this feature of F* is of particular importance because it allows to maintain compatibility between certified modules and those existing in the project which are not certified yet. We also discuss advantages and disadvantages of such approach and formulate topics for further research.

Keywords: certified programming; F*; program verification; context-free grammar; Chomsky normal form; grammar transformation; dependent type; refinement type

DOI: 10.15514/ISPRAS-2016-28(2)-8

For citation: Polubelova M.I., Bozhko S.N., Grigorev S.V. Certified Grammar Transformation to Chomsky Normal Form in F*. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 127-138. DOI: 10.15514/ISPRAS-2016-28(2)-8

1. Introduction

Certified programming is designed for proving that a program meets its specification. For this technique, proof assistants or interactive theorem prover are used [1], what allows to check correctness of the program at compile time and guarantees that the program always works according to its specification. Classical fields of application

of certified programming are the formalization of mathematics, security of cryptographic protocols and the certification of properties of programming languages.

There are two approaches to certified programming [2]. In the classical approach the program, its specification, and the proof that the program meets its specification are written separately, as different modules. Such technique costs too much to be applied in software development. More effective approach is to combine program, its specification, and the proof in one module by means of dependent types [3], [4]. The most well-known toolchains for program verification are Coq [5], Agda [6], F* [7] and Idris [8]. Among them, F* is the only language which supports semi-automated proving and general-purpose programming [9].

As a proof assistant, F* allows to formulate and prove properties of programs by using lemmas and enriching types. F* not only infers types of functions, but also the properties of its computations such as purity, statefulness, divergence. For example, consider the following function:

```
val f : (int -> Tot int) -> int -> Tot int
let f g x = g x
```

The keyword `val` indicates that we declare a function `f` and its type signature. The function `f` takes a function `g` and an integer value, as arguments. The effect of computation `Tot t` is used for total expression, which always evaluates to a `t`-typed result without entering an infinitive loop, throwing exception or other side effects. Hence, one can prove for some programs not only their properties and restrictions on the types, but also guarantee their termination and that a result has assigned type.

We apply certified programming using F* to a grammarware research and development project YaccConstructor (YC) [10], [11]. YC is a tool for parser construction and grammar processing. Also it is a framework for research and development of lexer and parser generators and other grammarware for .NET platform. The verification of its programs covers the topic of parser correctness: how to obtain formal evidence that a parser is correct with respect to its specification [12].

In this article, we consider only one algorithm implemented in YC, namely the transformation of context free grammar to Chomsky normal form, that is a small step towards the certification of entire project. The algorithm of grammar normalization consists of four transformations. We prove totality of each of them and establish an order of their application to the input grammar. In addition, we describe the peculiarities of evaluation F* as a proof assistant and formulate topics for further research.

2. Overview of F*

We use a functional programming language F* [7] for program verification. It is the only language that support semi-automated proving and general-purpose programming [9]. The main goal of this tool is to span the capabilities of interactive proof assistants like Coq [5] and Agda [6], general-purpose programming languages like OCaml and Haskell, and SMT-backed semi-automated program verification tools like Dafny [13] and WhyML [14].

Type system of F* includes polymorphism, dependent types, monadic effects, refinement types, and a weakest precondition calculus [15], [16]. These features allow expressing precise and compact specification for programs [7].

Dependent function type has the following form $x_1{:}t_1 \rightarrow .. \rightarrow x_n{:}t_n[x_1..x_{n-1}] \rightarrow E\ t\ [x_1..x_n]$. Each of a function's formal parameters are named x_i and each of these names in the scope to the right of the first arrow that follows it. The notation $t[x_1..x_m]$ indicates that the variables $x_1..x_m$ may appear free in t.

Refinement type has a form $x{:}t\{phi(x)\}$. It is a sub-type of t restricted to those expressions of type t that satisfy a predicate $phi(e)$.

In addition to inferring a type, F* also infers side effects of an expression such as exceptions and state. The following are the most significant monadic effects.

- `Tot t` – the effect of a computation that guarantees evaluation to a `t` - typed result, without entering an infinite loop, throwing an exception, reading or writing the program's state.
- `ML t` – the effect of a computation that may have arbitrary effects, but if some result is computed, then it is always of type `t`.
- `Dv t` – the effect of a computation that may diverge.
- `ST t` – the effect of a computation that may diverge, read, write or allocate on a heap.
- `Exn t` – the effect of a computation that may diverge or raise an exception.

The effects `{Tot, Dv, ST, Exn, ML}` are arranged in a lattice, with `Tot` at the bottom, `ML` at the top, and with `ST` unrelated to `Exn`.

There are two main approaches to prove properties: either by enriching the type of a function (intrinsic style) or by writing a separate lemma about it (extrinsic style). You can see an example of the first approach below; keyword `val` indicates declaration of a value and its type signature.

```
val append: l1:list 'a -> l2:list 'a
    -> Tot (l:list 'a{length l=length l1+length l2})
let rec append l1 l2 =
match l1 with
| [] -> l2
| hd :: tl -> hd :: append tl l2
```

The following example demonstrates extrinsic style, in which the formula after keyword `requires` is the pre-condition of the lemma, while the one after keyword `ensures` is its post-condition.

```
val append_len: l1:list 'a -> l2:list 'a
    -> Lemma (requires True)
       (ensures (length (append l1 l2)=
                 length l1 + length l2)))
let rec append_len l1 l2 =
```



```
match l1 with
| [] -> ()
| hd::tl  -> append_len tl l2
```

There is no general rule which style of proving to use, but in some cases it is impossible to prove a property of a function directly in its types and one has to use a lemma.

When defining lemmas or expressions that are total, F* automatically proves their termination. The termination check is based on a well-founded relation. For natural numbers, F* uses classical decreasing metric, for inductive types – the sub-term ordering, for recursive function, it requires the tuple of parameters to be in decreasing lexicographic ordering. The last case can be overridden with using clause `decreases` $\%[x_1..x_n]$, which explicitly chooses a lexicographic ordering on arguments.

To conclude, one can use F* to write effectful programs, specify them using dependent and refinement types, verify them using an SMT solver or providing interactive proofs. Programs written in F* can be translated to OCaml or F# for further execution.

3. Verification of transformation of CFG to CNF

In this section we briefly describe some necessary aspects of the theory of formal languages, sketch a totality proof for one of grammar transformations to Chomsky normal form in F*, and formulate some advantages and disadvantages of this approach.

3.1 Context-free grammar and Chomsky normal form

In this section we give basic definitions and formulate a theorem that helps us to verify the implemented algorithm of a transformation of context-free grammar to Chomsky normal form.

In formal language theory, a **context-free grammar** (CFG) is a formal grammar in which every production rule is of the form $A \rightarrow \alpha$, where A is single nonterminal symbol and α is a string of terminals and/or nonterminals (α can be empty).

Context-free grammar is said to be in **Chomsky normal form** (CNF) if all of its production rules are of the form:

- $A \rightarrow BC$
- $A \rightarrow a$
- $S \rightarrow \varepsilon$,

where A, B and C are nonterminal symbols, a is a terminal symbol, S is the start nonterminal, and ε denotes the empty string. Also, neither B nor C may be the start symbol, and the third production rule can only appear if ε is in $L(G)$, namely, the language produced by the context-free grammar G.

Context-free grammars given in Chomsky normal form are very convenient to use. It is often assumed that either CFGs are given in CNF from the beginning or there is an

intermediate step of normalization. Having a certified implementation of normalization for CFGs enables us to stop thinking in terms of CFG and consider grammar in CNF without losing guarantees of correctness.

CFG normalization theorem: There is an algorithm which converts any CFG into an equivalent one in Chomsky normal form.

The full normalization transformation for a CFG is a composition of the following constituent transformations.

- Replace all rules $A \to X_1 X_2 .. X_k$, where $k \geq 3$ with rules $A \to X_1 A_1$, $A_1 \to X_2 A_2$, .., $A_{k-2} \to X_{k-1} X_k$, where A_i are "fresh" nonterminals.
- Eliminate all ε-rules.
- Eliminate all chain rules.
- For each terminal a, add a new rule $A \to a$, where A is a "fresh" nonterminal and replacing a in the right-hand sides of all rules with length at least two with A.

3.2 Verification with F*

Our purpose is to verify a core YaccConstructor (YC) using F*. YC is an open source modular tool for research in lexical and syntax analysis and its main development language is F# [17]. In this paper we consider only a verification of normalization grammar algorithm [18] which is defined in a following way:

```
let toCNF (ruleList: Rule.t<_,_>list) =
    ruleList
      |> splitLongRules
      |> deleteEpsRules
      |> deleteChainRules
      |> renameTerm
```

The function `toCNF` is a composition of the four transformations mentioned. Notice that the order of rules execution is important. The first rule must be executed before the second, otherwise normalization time may increase to $O(2^n)$. The third rule follows the second, because elimination of ε-rules may produce new chain rules. Also, the fourth rule must be executed after the second and the third as they can generate useless symbols.

F*, as a proof assistant, allows to formulate and prove properties of function of interest using lemmas or enriching types. For example, in F# function `(f (x:int)= 2*x)` is inferred to have type `(int -> int)`, while in F* we infer `(int -> Tot int)`. This indicates that `(f (x:int) = 2*x)` is a pure total function which always evaluates to `int`. A lemma is a ghost total function that always returns the single unit value `()`. When we specify a total function, we have to prove totality of every nested function, because F* supports only high-level annotations. In others words, we cannot add annotation for a nested function. Therefore, to prove totality of a function

containing nested functions, we need to lift all nested functions up and explicitly prove totality of these functions.

We describe each function of interest in an individual module to avoid namespace collision. We use module architecture similar to YC architecture. Module `IL` contains type constructors for describing productions of a grammar. Module `Namer` contains a function to generate new names. Finally, we created individual modules for each transformation and a separate, main, module which contains the definition of `toCNF` transformation.

We implemented all the transformations in F* [19], but in this paper we consider only one of them, namely `SplitLongRules`, which eliminates long rules. Firstly, we describe all the helpers we need, prove their totality and other necessary properties, and then explain why this transformation is correct.

In the first transformation, it is necessary to create new nonterminals, so we need a function to supply them. The function `Namer.newSource` defined below is used.

```
val newSource: n:int -> oldSource:Source -> Tot Source
let newSource n old =
       ({old with text = old.text^(string_of_int n)})
```

Integer `n` is equal to the size of the list of rules which we have at the moment of function `Namer.newSource` call. Obviously, function `Namer.newSource` is injective. In other words, unique rule names remain unique after application `splitLongRules`.

Some necessary helpers are grouped in `TransformAux` module: for example, functions `createRule` and `createDefaultElem`, which take some arguments and return `Rule` and `Elem` respectively. `Elem` is the right part of the rule if the latter is a sequence. Also, we define follow one simple function which returns the length of the right part of the rule.

```
val lengthBodyRule: Rule 'a 'b -> Tot int
let lengthBodyRule rule =
       List.length (match rule.body with
                    |PSeq(e, a, l) -> e
                    | _ -> [ ])
```

The most interesting function is `cutRule`. It takes a rule and a list-accumulator as an input. If the length of the right-hand side of the rule is less or equal to 2, `cutRule` only renames a nonterminal to avoid name collision. Otherwise, it is necessary to create new nonterminal B_{k-2}, cut off last two elements $X_{k-1} X_k$, pack them into a new rule $B_{k-2} \to X_{k-1} X_k$, and then add the nonterminal to the end of the long rule. Then the new rule is added to the accumulator and the function `cutRule` is recursively called on the new rule and accumulator. This way, we reduce our rule by one. Function signature is the following.

```
val cutRule: rule:(Rule 'a 'b)
    -> resRuleList:(list (Rule 'a 'b))
    -> Tot (list (Rule 'a 'b))
```

```
        (decreases %[lengthBodyRule rule])
```

There are some peculiarities in our implementation, which are worth mentioning. One of them is the representation of the right-hand side of the rules by lists. In the algorithm, we need to cut off two last elements of a rule, so we carry out the following steps.

```
let revEls = List.rev elements
let cutOffEls = [List.Tot.hd revEls;
                 List.Tot.hd (List.Tot.tl revEls)]
```

Functions `List.hd` and `List.tl` from a standard library are not defined for an empty list, so they cannot be considered total, which limits their usage in our code. In F* there is a module `List.Tot` which provide proper total analogs of the functions mentioned. We only provide their signature here.

```
val hd: l:list 'a{is_Cons l} -> Tot 'a
val tl: l:list 'a{is_Cons l} -> Tot (list 'a)
```

Predicate `is_Cons` takes a list as an input and returns `false` if it is empty, otherwise it returns `true`.

If function `List.Tot.hd` is applied to a list, nonemptiness of which is not clear from the context, F* reports a type mismatch. A pleasant peculiarity of F* is that in some rare cases it can derive necessary properties. In our implementation of the transformation, only the rules which have more than two symbols in the right-hand side are split. In this case F* is able to automatically derive required type, so we can choose two elements. It can be illustrated with the following example.

```
// lst has type list int and can be empty
assume val lst: list int
// f takes only nonempty lists
assume val f: lst:(list int){is_Cons lst} -> Tot int
assume val g: lst:(list int) -> Tot int

//Ok
let test1 (lst:list int) =
    if List.length lst >= 1
    then f lst
    else g lst

//Fail: subtyping check failed
let test2 (lst:list int) =
    if List.length lst >= 0
    then f lst
    else g lst
```

At the same time, we have to prove and explicitly add even simple lemmas for functions. For example, if list `lst` has type `(list 'a){is_Cons lst}`, then F* can only infer that `(List.rev lst)` has type `(list 'a)`. This can be easily fixed

with the instruction `SMTPat`. In addition, we should formulate the following lemma which proof can be derived automatically by F*. The following code makes `List.rev` preserve information about the length:

```
val rev_length: l:(list 'a)
    -> Lemma (requires true)
    (ensures (List.length (List.rev l) = List.length l))
    [SMTPat (List.rev l)]
```

We proved totality of all the nested functions. Now we want to prove termination of the general one. In our case, it is sufficient that the length of the rule strictly decreases on each recursive call and we are not interested in the length of the accumulator. To prove this we must explicitly specify that after applying `List.Tot.tl` to a list, its length reduces by 1. So, we must use the same method as we used before.

```
val tail_length : l:(list 'a){is_Cons l}
-> Lemma (requires True)
(ensures(List.length (List.Tot.tl l)=(List.length l)-1))
[SMTPat (List.Tot.tl l)]
```

With this sufficient information F* has to conclude that `cutRule` is total.

Function `splitLongRules` takes a list of rules and applies `cutRule` to each rule, then concatenates all the results and returns the combined list.

```
val splitLongRules: list (Rule 'a 'b)
    -> Tot (list (Rule 'a 'b))
let splitLongRules ruleList =
    List.Tot.collect
       (fun rule -> cutRule rule [ ]) ruleList
```

Totality is proved automatically by F*.

Previously we proved totality of our transformation, but we had not mentioned properties of the rules we get after applying `splitLongRules`. We add restriction on the type of function, which guarantees the necessary property of the result, instead of proving the lemma about these properties. The function signature now look like this.

```
val cutRule: rule:(Rule 'a 'b)
    -> acc:(list (Rule 'a  'b))
   {List.Tot.for_all (fun x->lengthBodyRule x<=2) acc}
    -> Tot (res:(list (Rule 'a 'b))
   {List.Tot.for_all (fun x->lengthBodyRule x<=2) res})
   (decreases %[lengthBodyRule rule])

val splitLongRules:list (Rule 'a 'b)
    -> Tot (res:list (Rule 'a 'b)
   {List.Tot.for_all (fun x->lengthBodyRule x<=2) res})
```

Now we have almost everything we need to prove such properties. We have to provide some additional information so that F* could check arguments type when `collect`

is recursively called. At the moment of cutting the rule off, we should fix the length in the type of the cut part. For this purpose we have to define a function to take our list and return part with that type. Further, we have to prove lemma that states that concatenation of two lists with short rules is the list with short rules. After that F* accepts type correctness.

3.3 Advantages and disadvantages of F*

In this section we want to outline some advantages and disadvantages of F* programming language. In F#, even if there is no doubt that some functions are correct, an incorrect result may still be obtained by applying them in a wrong order. F* can prevent such situations, if a programmer specifies the properties demanded from an input data in a function type. For instance, `deleteChainRules` should only be applied after deleting epsilon rules. This can be ensured by specifying the following signature of `deleteChainRules` function (where predicate `has_no_eps_rules` checks that there are no epsilon rules).

```
val deleteChainRules:
    ruleList:(list (Rule 'a 'b))
    {has_no_eps_rules ruleList}
    -> Tot (list (Rule 'a 'b))
```

Unfortunately, there are some disadvantages of F* which we want to emphasize. First of all, it does not provide any – even primitive – support for object-oriented features. One can use structures instead of classes, but it complicates development. For example, we had to explicitly create functions for constructing elements of types. In other words, rather than create class `Person` with constructors and methods:

```
let person = new Person("Nick", 27)
```

One has to write code in a rather cumbersome manner:

```
let new_Person name age = {name=name; age=age}
```

There is a special construct in many functional languages which checks whether some property holds for a value. Such construct is called `guard` in Haskell and `when` in OCaml and F# and is often used in pattern matching to simplify code. Unfortunately, it is not supported in F* and one can only hope that it will be supported in the latter language versions.

Lastly, we can notice poor quality of error reporting in F* which sometimes makes it hard to understand why proofs do not pass correctness tests.

4. Conclusion and future work

We presented a verification of one of transformations of context-free grammar to the Chomsky normal form. We proved totality of each function implemented, as this property guarantees that computations always terminate and do not have side effects, which is useful in practice. Although for a complete proof of the correctness of the grammar transformation we still need to prove the equivalence of the original and

the resulting grammar, we have already obtained interesting results. We can specify an input and an output of functions – using refinement and dependent types – that allows us to establish application order of the four transformations, by means of which correctness of the whole transformation is guaranteed.

We use programming language F* to verify the implementation, but to be able to execute it one needs to extract it to OCaml or F# and then compile it using the OCaml or F# compiler respectively. At the moment, the mechanism of extraction code from F* to F# omits casts, erases dependent types, higher rank polymorphism and ghost computation [9]. These features are very important and lack of them breaks the consistency and correctness of programs within the target language. F* is currently under active development, and implementation of the extraction mechanism which copes with the above shortcoming is actual topic of our further research.

References

[1]. H. Geuvers. Proof assistants: History, ideas and future. Sadhana, 2009, vol. 34, issue 1. pp. 3–25. DOI: 10.1007/s12046-009-0001-5.

[2]. A. Chlipala. Certified programming with dependent types. MIT Press, 2013, 440 p.

[3]. T. Sheard, A. Stump, S. Weirich. Language-based verification will change the world. Proceedings of the FSE/SDP workshop on Future of software engineering research, 2010, pp. 343–348. DOI: 10.1145/1882362.1882432.

[4]. E. Tanter, N. Tabareau. Gradual certified programming in Coq. Proceedings of the 11th Symposium on Dynamic Languages, 2015. pp. 26–40. DOI: 10.1145/2816707.2816710.

[5]. The Coq proof assistant. June 2016. https://coq.inria.fr/

[6]. U. Norell. Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

[7]. The F* homepage. June 2016. https://www.fstar-lang.org/

[8]. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming, 2013, vol. 23, n. 5, pp. 552–593. DOI: 10.1017/S095679681300018X.

[9]. N. Swamy, C. Hritcu, C. Keller. Dependent Types and Multi-Monadic Effects in F*. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655

[10]. The YaccConstructor homepage. June 2016. https://github.com/YaccConstructor/

[11]. I. Kirilenko, S. Grigorev, D. Avdiukhin. Syntax analyzers development in automated reengineering of informational system. St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems, 2013, no. 174, pp. 94 – 98.

[12]. J.-H. Jourdan, F. Pottier, X. Leroy. Validating LR (1) parsers. Programming Languages and Systems, 2012, pp. 397–416.

[13]. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. International Conference on Logic for Programming Artificial Intelligence and Reasoning, 2010, pp. 348–370.

[14]. J.-C. Filliatre, A. Paskevich. Why3: Where Programs Meet Provers. Proceedings of the 22Nd European Conference on Programming Languages and Systems, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8.

[15]. B. C. Pierce. Types and Programming Languages. MIT Press, 2002, 645 p.
[16]. F* tutorial. June 2016. https://www.fstar-lang.org/tutorial/
[17]. D. Syme, A. Granicz, A. Cisternino. Expert F#. Apress, 2012, 609 p.
[18]. D. Firsov, T. Uustalu. Certified normalization of context-free grammars. Proceedings of the 2015 Conference on Certified Programs and Proofs, 2015, pp. 167–174. DOI: 10.1145/2676724.2693177.
[19]. Verification of grammar transformation to Chomsky normal form in F*. June 2016. https://github.com/YaccConstructor/YC_FStar.

Верификация преобразования грамматики в нормальную форму Хомского в F*

М.И. Полубелова <polubelovam@gmail.com>
С.Н. Божко <gkerfimf@gmail.com>
С.В. Григорьев <Semen.Grigorev@jetbrains.com>
Санкт-Петербургский государственный университет,
199034, Россия, Санкт-Петербург, Университетская наб., д.7/9

Аннотация. Сертификационное программирование позволяет доказывать, что программа соответствует своему формальному описанию. Проверка корректности производится статически, благодаря чему становится возможным отказаться от дальнейшего тестирования проверифицированных программ. Среди инструментов, предназначенных для сертификационного программирования, только инструмент F* позволяет реализовывать программы на языке общего назначения и автоматизирует доказательство их корректности. Последнее означает, что инструмент F* автоматически выведет доказательство корректности, где это возможно, при этом пользователь может специфицировать более сложные доказательства, если это необходимо. Мы работаем над применением данного подхода к проекту YaccConstructor – платформе для исследования и разработки генераторов лексических и синтаксических анализаторов и других алгоритмов для работы с грамматиками. В данной статье рассматривается верификация реализации одного из таких алгоритмов – преобразования грамматики в нормальную форму Хомского – что является первой задачей на пути к верификации всего проекта YaccConstructor. Для программы, реализующей данное преобразование, доказаны завершаемость и тотальность, а также установлен порядок применения используемых в ней основных преобразований с использованием зависимых и уточняющих типов. Следующим важным направлением данной работы является доказательство эквивалентности исходной и преобразованной грамматик. Инструмент F* позволяет извлекать код, написанный на F*, как программу на языке программирования F# или OCaml. Так как F# является основным языком разработки проекта YaccConstructor, это позволит сохранить совместимость проверифицированных программ с существующими в проекте. В статье сформулированы преимущества и недостатки применения инструмента F*.

Ключевые слова: сертификационное программирование; F*; верификация программ; контекстно-свободная грамматика; нормальная форма Хомского; преобразование грамматики; dependent type; refinement type.

DOI: 10.15514/ISPRAS-2016-28(2)-8

Для цитирования: Полубелова М.И., Божко С.Н., Григорьев С.В. Верификация преобразования грамматики в нормальную форму Хомского в F*. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 127-138 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-8

Список литературы

[1]. H. Geuvers. Proof assistants: History, ideas and future. Sadhana, 2009, vol. 34, issue 1. pp. 3–25. DOI: 10.1007/s12046-009-0001-5.
[2]. A. Chlipala. Certified programming with dependent types. MIT Press, 2013, 440 p.
[3]. T. Sheard, A. Stump, S. Weirich. Language-based verification will change the world. Proceedings of the FSE/SDP workshop on Future of software engineering research, 2010, pp. 343–348. DOI: 10.1145/1882362.1882432.
[4]. E. Tanter, N. Tabareau. Gradual certified programming in Coq. Proceedings of the 11th Symposium on Dynamic Languages, 2015. pp. 26–40. DOI: 10.1145/2816707.2816710.
[5]. The Coq proof assistant. June 2016. https://coq.inria.fr/
[6]. U. Norell. Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.
[7]. The F* homepage. June 2016. https://www.fstar-lang.org/
[8]. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming, 2013, vol. 23, n. 5, pp. 552–593. DOI: 10.1017/S095679681300018X.
[9]. N. Swamy, C. Hritcu, C. Keller. Dependent Types and Multi-Monadic Effects in F*. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655
[10]. The YaccConstructor homepage. June 2016. https://github.com/YaccConstructor/
[11]. I. Kirilenko, S. Grigorev, D. Avdiukhin. Syntax analyzers development in automated reengineering of informational system. St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems, 2013, no. 174, pp. 94 – 98.
[12]. J.-H. Jourdan, F. Pottier, X. Leroy. Validating LR (1) parsers. Programming Languages and Systems, 2012, pp. 397–416.
[13]. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. International Conference on Logic for Programming Artificial Intelligence and Reasoning, 2010, pp. 348–370.
[14]. J.-C. Filliatre, A. Paskevich. Why3: Where Programs Meet Provers. Proceedings of the 22Nd European Conference on Programming Languages and Systems, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8.
[15]. B. C. Pierce. Types and Programming Languages. MIT Press, 2002, 645 p.
[16]. F* tutorial. June 2016. https://www.fstar-lang.org/tutorial/
[17]. D. Syme, A. Granicz, A. Cisternino. Expert F#. Apress, 2012, 609 p.
[18]. D. Firsov, T. Uustalu. Certified normalization of context-free grammars. Proceedings of the 2015 Conference on Certified Programs and Proofs, 2015, pp. 167–174. DOI: 10.1145/2676724.2693177.
[19]. Verification of grammar transformation to Chomsky normal form in F*. June 2016. https://github.com/YaccConstructor/YC_FStar.