Regular Paper

Safe Low-level Code Generation in Coq Using Monomorphization and Monadification

Akira Tanaka[1,a) Reynald Affeldt[1,b) Jacques Garrigue[2,c)

Received: April 8, 2017, Accepted: August 9, 2017

Abstract: Our goal is the production of formally-verified pieces of low-level code. Low-level code is typically written in C, so as to enable efficient manipulation of data at the bit-level and easy access to built-in features of CPUs. Proof-assistants arguably provide the most rigorous approach to formal verification of computer programs. Unfortunately, they only allow for extraction of runnable code in high-level languages such as ML. Of course it is possible to embed C snippets into ML programs, but this results in a complicated extraction process and the performance of the output program becomes difficult to anticipate. In this paper, we propose a new code generation scheme for the Coq proof-assistant that directly generates provably-safe C code. It is implemented in the form of plugins. The generation of C source code is done by a plugin performing beforehand monomorphization of Coq programs. The correctness of monomorphization can be proved within Coq. Code generation allows for user-guided changes of data structures. It is therefore possible to do formal verification using proof-friendly data structures, while enjoying optimized C representations in the output code. In order to ensure the safety of this transformation, we propose a new customizable monadification algorithm in the form of another plugin. Using monadification, one can ensure by the insertion of the right monads the preservation of critical invariants, such as the absence of overflows or complexity properties. We provide several examples to illustrate our approach, including a realistic use-case: the rank algorithm from succinct data structures.

Keywords: Coq, C, monomorphization, monadification, code generation

1. Introduction

1.1 Main Motivation

In this research, we address the general topic of producing formally verified pieces of critical code. Our original and main motivation [26] is the formal verification of code for succinct data structures. These are data structures for big data analysis that use a minimal amount of computer memory [19]. While we will see that our results are not restricted to this sole application, we will use it as an example for the sake of concreteness. Succinct data structures are typically written in C (or C++) because they manipulate data at the bit-level. In particular, with succinct data structures, one typically wants to take advantage of built-in functions for features of recent CPUs (e.g., `__builtin_popcount`). However, proof-assistants used for formal verification typically only allow for extraction of code in higher-level languages such as OCaml. This is why, in our previous work [26], we resorted to embedding of C snippets into OCaml code. However, this resulted in an arguably involved process for code extraction.

1.2 Problems with the Standard Approach to Code Extraction

Taking a step back, we came to realize that, in the context of our research, the Coq extraction [13] facility suffers many defects and prevents improvements. Here are the main issues that we have identified:

- <u>Efficiency</u> It is not easy to use native features of processors such as 64bit integers, SSE, `__builtin_popcount`, etc.
- <u>Robustness</u> Extraction is not robust. It is too sensitive to customization (Appendix A.1 provides an example in which customization causes the code generator to fail to preserve tail recursion). There are some high-level options to mitigate inefficiencies (for example, to enforce selective inlining with the `Extraction Inline` command) but they do not address the unpredictability of the code generator. To achieve robustness, we believe that it is important to improve flexibility so as to be able to obtain carefully-crafted code.
- <u>Correctness</u> In practice, Coq natural numbers are often extracted to OCaml integers but this is unsafe in general because the range of the latter is limited. There is no generic mechanism to guarantee the safety of such a translation.
- <u>Modularity</u> The implementation of the extraction facility should be more modular. Since it is part of the trusted base, an implementation with well-identified modules is obviously desirable. In its current form, the extraction facility does not lend itself well to modifications and extensions. Indeed, modified extraction with new language support requires the user to build and install modified versions of Coq (see the many such back-ends in Section 8.3).
- <u>Type-specificity</u> Low-level C uses types to optimize the representation of values. However, Coq extraction attempts to translate all of Gallina (the language of the Coq proof assis-

1 National Institute of Advanced Industrial Science and Technology (AIST), AIST Tsukuba Central 1, Tsukuba, Ibaraki 305–8560, Japan
2 Graduate School of Mathematics, Nagoya University, Nagoya, Aichi 464–8602, Japan
a) tanaka-akira@aist.go.jp
b) reynald.affeldt@aist.go.jp
c) garrigue@math.nagoya-u.ac.jp

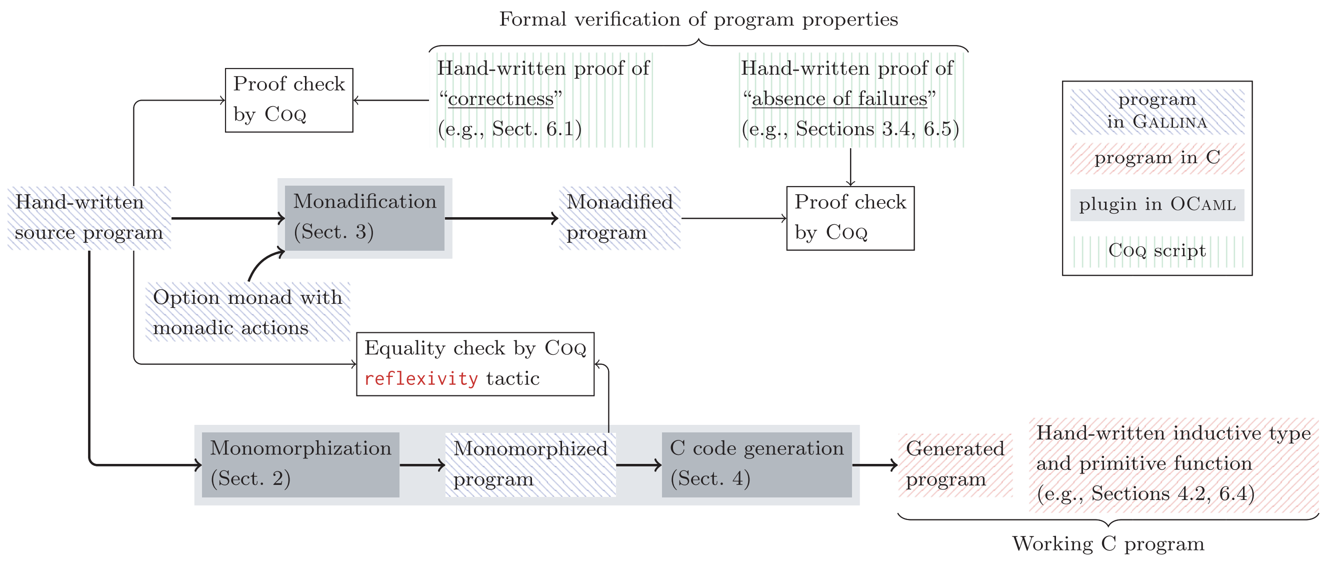
Formal verification of program properties
Proof check by Coq
Hand-written proof of "correctness" (e.g., Sect. 6.1)
Hand-written proof of "absence of failures" (e.g., Sections 3.4, 6.5)
program in Gallina
program in C
plugin in OCaml
Coq script
Hand-written source program
Monadification (Sect. 3)
Monadified program
Proof check by Coq
Option monad with monadic actions
Equality check by Coq
reflexivity tactic
Monomorphization (Sect. 2)
Monomorphized program
C code generation (Sect. 4)
Generated program
Hand-written inductive type and primitive function (e.g., Sections 4.2, 6.4)
Working C program

Fig. 1 The structure of our verification and code generation scheme.

tant), including polymorphism and dependent types, and as a result only supports uniform translation. This prevents the generation of efficient C code, and does not allow to change the translation of a function according to its type.

1.3 Our Approach to Low-level Code Generation

To overcome the issues explained in Section 1.2, we propose a new approach for the generation of efficient code from a Coq program. Here are our main design decisions:

- C Code Generation Use C as the output language so as to be able to use easily the processors' native features and `goto` statements to achieve tail recursion (this addresses the Efficiency and Robustness issues). Indeed, in our case, there is no strong reason to use the OCaml code generated by the default extraction facility of Coq.
- Monomorphization Use *monomorphization* to specialize polymorphic functions and avoid uniform translation (this addresses the Type-specificity issue).
- Monadification Use *monadification* to be able to assert the correctness of the change of data structures (e.g., from Coq natural numbers to C's `int`s). This provides a way to assess the safety of the extraction (this addresses the Correctness issue). See Section 1.4 just below for background information about monadification for Gallina.
- Plugins Use plugins to build our new code generation scheme (this addresses the Modularity issue). Indeed, there is no strong reason to try to reuse or extend the current Coq extraction facility. One could think of adding a new "MiniML-to-C" layer to the latter but this is made complicated by the fact that MiniML lacks explicit type information (the difficulties caused by this lack of type information are illustrated in Ref. [9]). Concretely, we provide one plugin for monomorphization and C code generation, and one plugin for monadification. This has many benefits. The correctness of the monomorphization plugin can be asserted independently inside Coq. At the time of this writing, the C code generation is trusted but small because it does not (need to) cover the full Gallina language (for example, there is no need to handle polymorphism that C lacks). On the practical side, plugins can be installed individually and used with a standard Coq installation, possibly for another purpose.

Figure 1 summarizes our approach by showing how we generate a working C program from a Gallina program and how we formally establish its properties.

1.4 Monadification for Coq

The basic idea of Monadification is to see a Gallina term not as a pure lambda-term, but as a possibly effectful program written in a strict functional programming language, such as ML. This is done by transforming the original term into a new Gallina term, where all effects are encapsulated in a monad. The choice of the specific monad, and in particular which constructions are to be interpreted as monadic actions, depends on the effect we want to consider. This means parameterizing on the monadic triple, which is a parameterized type "M" together with two functions "$\mathsf{bind} : \forall ab, M\ a \to (a \to M\ b) \to M\ b$" and "$\mathsf{return} : \forall a, a \to M\ a$", while selectively converting some specific functions into monadic actions, causing effects in the monad. By observing these effects, it becomes possible to prove that invariants required by our translation scheme (into C) are indeed enforced. See Section 3.4 for basic examples and Section 6.5 for a realistic application.

In the practical context of code generation, it is not reasonable to perform monadification manually. Manual monadification is obviously error-prone. It is also repetitive, especially when one needs to monadify the same program with different monads (for example, when one needs a monad to prove that the program never fails and another to establish complexity). Last, it is a daunting task; for example, when we want to introduce a monadic action for a very basic function, such as S (the Coq successor function for natural numbers), it forces us to monadify not only the target program, but also large parts of the Coq standard library.

In this work, we introduce a new monadification algorithm and apply it to Gallina. We explain the basic idea of this algorithm in Section 3.1 and provide a detailed description in Section 3.2. Although automatic monadification itself is not a new idea, existing approaches turn out to be unsatisfactory for the purpose of generating C code from Gallina; this is discussed in Section 8.5.

1.5 Outline of This Paper

This paper is organized as follows.

We first explain how we carry out the generation of C code in Sections 2–4. In Section 2, we explain the monomorphization plugin. In Section 3, we explain the monadification plugin. In Section 4, we explain the generation of C code.

Then, we perform several experiments to evaluate our scheme for the generation of C code in Sections 5–7. In Section 5, we test our monadification plugin using the SSReflect implementation of lists. In Section 6, we demonstrate our approach to C code generation with the example of the *rank* algorithm for succinct data structures. In Section 7, we evaluate more precisely the size of the trusted base our approach introduces.

Finally, we review related work in Section 8 and conclude in Section 9.

2. Monomorphization

Monomorphization is the action of specializing (polymorphic) definitions w.r.t. type arguments. It is an important step towards generation of C code, which does not enjoy polymorphism. We have implemented monomorphization as a Coq plugin. When its input is a non-recursive function (i.e., `fun` and `Definition` constructs), the correctness of the output can be checked automatically inside Coq using the `reflexivity` tactic. We illustrate monomorphization with a simple example in Section 2.1 and further discuss design decisions in Sections 2.2–2.6.

2.1 Simple Example of Monomorphization

We illustrate our monomorphization plugin with the example of a polymorphic definition of pair-swapping [*1]:

```
Definition swap {A B : Type} (p : A * B) :=
 let (a, b) := p in (b, a).
Definition swap_bb p := @swap bool bool p.
```

The `Monomorphization` command monomorphizes a function as well as related data structures. For example, monomorphizing `swap_bb` also monomorphizes the `swap` function and the `pair` constructor (i.e., (_, _)) (see Section 2.3 for details). Concretely, the command

```
Monomorphization swap_bb.
```

outputs the following monomorphized definitions:

```
Definition _pair_bool_bool :=
 @pair bool bool : bool → bool → bool * bool.
Definition _swap_bool_bool (p : bool * bool) :=
 let (a, b) := p in _pair_bool_bool b a.
Definition _swap_bb p :=
 _swap_bool_bool p : bool * bool → bool * bool.
```

*1 `@` is used in Coq to avoid automatic filling of implicit arguments.

We can make sure that the semantics of the resulting function has not been altered by proving equality between the polymorphic and the monomorphized definitions:

```
Goal swap_bb = _swap_bb. Proof. reflexivity. Qed.
```

In this proof, `reflexivity` checks that `swap_bb` and `_swap_bb` are convertible. It succeeds because the proof amounts to β-reduction which is part of the conversion rule in Coq [*2].

2.2 Overview of the Monomorphization Process

Let us describe informally the process of (function) monomorphization as implemented by our plugin. The main command is:

`Monomorphization` *function_name*$_0$ *function_name*$_1$ ...

It triggers traversal in depth-first order of the functions given in input. Type arguments are discovered at application sites (so, in particular, the functions given in input should not have type arguments). More precisely, when `Monomorphization` finds a function with its actual type arguments, it defines a function specialized w.r.t. these arguments and the function is examined to find further function/constructor invocations.

Constructors of inductive types are treated similarly to functions. They are monomorphized and specialized functions to invoke the constructors with type arguments are defined (as `_pair_bool_bool` in the above example).

Not only global definitions, but also local definitions by `let` expressions are the target of monomorphization. Such definitions are copied for each actual type arguments and nested `let`s are generated. For illustration, let us consider the following function:

```
Definition foo :=
 let id := (fun (T : Type) (x : T) ⇒ x) in
 (id bool true, id nat 0).
```

Monomorphization of `foo` causes the locally-defined polymorphic function `id` to be monomorphized into `id` and `id0`:

```
Definition _foo :=
 let id := fun x : bool ⇒ x in
 let id0 := fun x : nat ⇒ x in
 let b := _true in
 let b0 := id b in
 let n := _O in
 let n0 := id0 n in
 _pair_bool_nat b0 n0
```

2.3 User Control of the Program Traversal

As seen in the example in Section 2.1, monomorphization proceeds by specializing definitions. However, not all definitions are worth specializing and some are not even traversable. In order to control the program traversal, our plugin provides the following command:

`Terminate Monomorphization` *function type_arguments* ...

For example, one does not need to specialize functions

*2 The case of recursive functions cannot be as easily automated because the conversion rule does not handle the `fix` and `Fixpoint` constructs.

that map directly to C instructions. This is the case of the function `addn` that performs addition of two natural numbers: it directly maps to the + operator in C. One can prevent its monomorphization by executing the command `Terminate Monomorphization addn`.

Functions may use opaque definitions that are not traversable at all. For example, the function `divn` that performs division of two natural numbers rely on the boolean equality between natural numbers. In SSReflect, this equality is defined by the expression `@eq_op nat_eqType a b` (hidden by the overloaded notation ==) where `nat_eqType` contains `eqnP` which is opaque. One can prevent the monomorphization of `divn` with the command `Terminate Monomorphization divn`.

2.4 Insertions of `let` Expressions

We chose to have the `Monomorphization` command insert local variables through `let` expressions. The purpose is to avoid nested expressions to facilitate code generation.

This transformation is perfectly safe: since the Coq conversion rule reduces `let` expressions (this is called ζ-reduction), the correctness can be proved using the `reflexivity` tactic. The result is similar to A-normal form [6]. A-normal form restricts nested expressions to constant, variable or lambda expression. We restricted nested expression only to variable since constant and lambda expression are complex operations for our C code generator: a constant is compiled to zero-argument function call and a lambda expression needs closure allocation (closure is not supported yet, see Section 9.1 for more insights).

The example of the `rank_init` function below illustrates the insertion of `let` expression. The function contains an expression `let w1 := neq0 (bitlen (n %/ sz1 * sz1)) in` ... and it is translated with the additional variables `n0`, `n1` and `n2`:

```
Definition _rank_init (b : bool) (s : bits) :=
 ...
 let n0 := _divn n sz1 in
 let n1 := _muln n0 sz1 in
 let n2 := _bitlen n1 in
 let w1 := _neq0 n2 in
 ...
```

2.5 About the Supported Subset of Gallina

Our monomorphization plugin does not need to support the full Gallina language. Indeed, we want to generate C code from resembling Gallina programs; there is therefore no need to support all the abstraction features of Gallina. Concretely, we require the user to write programs in the "ML-polymorphic subset" of Gallina. This can be achieved by abiding to the following rules:

- Type arguments must be made explicit. (This is required by Gallina.)
- A polymorphic function must be defined by a let-binding (or by a global definition) such as `let f (T : Type) x :=` ... `in` .... Type arguments must be placed in leading positions; for example, `fun (x : nat) (T : Type) ⇒` ... is prohibited.
- Polymorphic functions must be instantiated to be monomorphic, i.e., all occurrences of a polymorphic function must be applied to type arguments immediately. For example, if `f` is defined by `let f (T : Type) x :=` ... `in` ..., it must appear as `f t`. First-class polymorphism is not supported.
- Functions with polymorphic recursion are not supported. Recursive function application cannot change the type of arguments. For example, `t` must be `T` itself in `Fixpoint f (T : Type) x :=` ... `(f t y)` ....
- As a consequence, recursive inductive types must not change the type of their arguments. For example, `t` must be `T` itself in `Inductive I (T : Type) :=` ... `| C :` ... `→ I t →` ... `→ I T |` .... This is allowed in SML and supported by the monomorphization of MLton [15], but is of little use without polymorphic recursion.

2.6 Naming Rule for Monomorphization

We conclude this section with the naming rules for monomorphized functions. We chose rules such that the generated code is still readable.

Concretely, we use the original name (without module names), prefixed with an underscore, and append underscore-prefixed type arguments. For example, when `Coq.Init.Datatypes.pair` of type `∀ A B : Type, A → B → A * B` (this is the example from Section 2.1) is specialized for two `bool`s, it generates the monomorphized function with name `_pair_bool_bool`.

The naming rules defined above require in particular that complex types be turned into names. If a type is an inductive type without parameters, the name is unchanged (e.g., `bool`). If a type is an inductive type with parameters, such as $I\ t_1\ \dots\ t_n$, the parameter types are translated recursively and they are concatenated as $I\_t_1\_\dots\_t_n$. If a type is a function type, such as $t_1 \rightarrow t_2$, t_1 and t_2 are translated recursively and they are concatenated as t_1`_to_`t_2.

3. Monadification

In this section, we explain how we achieve Monadification for Coq as briefly explained in the introduction (Section 1.4).

We start by explaining the basic idea of our monadification algorithm in Section 3.1. We then explain our monadification algorithm in details in Section 3.2. We explain the implementation of this algorithm as a Coq plugin in Section 3.3. Finally, we provide concrete examples of monadification as performed by our plugin in Section 3.4: application of monadification to the detection of integer overflow (Section 3.4.1), buffer overrun (Section 3.4.2), and proof about complexity (Section 3.4.3).

Comparison with related work, including why they fail to fulfill our purpose, is deferred to Section 8.5.

3.1 The Idea of the Monadification Algorithm

When we implement Coq `nat` using C's `int`, we would like to prove that integer overflow never happens. However we cannot prove that $n + 1 - 1$ may overflow since $n + 1 - 1$ and n are equivalent in Coq. We use monadification and the option monad to make such program failures provable. Unless stated otherwise [*3], we will be using the option monad, which captures program failures: the `Some` constructor represents success, while the `None`

[*3] Section 3.4.3 defines another monad to count constructor calls for example.

constructor represents failure [*4]:

```
Definition ret {A} (x : A) := Some x.
Definition bind {A} {B}
  (x' : option A) (f : A → option B) :=
 match x' with
 | None ⇒ None
 | Some x ⇒ f x
 end.
```

We introduce Haskell-like notations for the monad:

```
Notation "'return' t" := (ret t) (at level 100).
Notation "x ≫= y" := (bind x y)
 (at level 65, left associativity).
```

When we show that a function `f x` never fails, we prove ∀`x`, *condition* → `fM x = Some (f x)` where `fM` is the monadified function of `f`. The monadification maps some specific functions whose behavior differs in Coq and in the target language, such as addition in C, to monadic actions which return `None` when the target language behavior may differ. *condition* is a condition sufficient for `f x` to have the same behavior in Coq and in the target language.

As this proposition is easier to prove if `f` and `fM` are similar, we would like to modify the code as little as possible, i.e. to introduce as few `bind` and `ret` operations as possible. Every time we apply a function whose result is in the monad, we need to insert a `bind` operation in the code, so we want to avoid introducing too many monadic constructors while monadifying.

Our main target is C programs. A multiple argument C function has no effect before the last argument is given. So, a two-arguments function, `f` : $t_1 \to t_2 \to t$, can be monadified to represent effects as `fM` : $t_1 \to t_2 \to M\ t$, or `f` itself if `f` has no effect. Inserting the monad for all subexpressions, `fM'` : $M\ (t_1 \to M\ (t_2 \to M\ t))$, is not required. `fM` needs less `bind`'s in its application than `fM'`. `fM` is invoked as `fM a b` but `fM'` is invoked as `fM' ≫= fun g ⇒ g a ≫= fun h ⇒ h b`.

We generalized this idea. Our monadification algorithm tries to insert as few monads as possible: `f` : $t_1 \to \ldots \to t_k \to t_{k+1} \to \ldots \to t_n$ is transformed to `fM` : $t_1 \to \ldots \to t_k \to M\ (t_{k+1} \to M\ (\ldots \to M\ (t_{n-1} \to M\ t_n)\ldots))$ for some k, chosen as big as allowed. We call k the *impure arity* of `f` since it is the number of arguments before the first effect occurs. Also, we consider the impure arity of a pure function as the arity of the pure function plus one. The monadification algorithm propagates impure arities and transforms expressions.

In particular, when it detects that a function has no effect (i.e., no monadic action), the function is kept as it is, without defining any monadified version. In such a case, we do not need to prove the equality between the original definition and the monadified definition.

3.2 Algorithm

In this section we present the concrete monadification algorithm. It takes as input a Coq program, composed of functions

[*4] This monad corresponds to the partiality monad in Ref. [16].

T	:	type variable
I	:	inductive type name
t	::=	$T \mid I\ t \ldots t \mid t \to t \mid sort$
x	:	variable
c	:	constant
C	:	constructor
k	:	natural number
v	::=	$t \mid x \mid c \mid C \mid f$
f	::=	$\mathsf{fun}\ v : t \Rightarrow e \mid \mathsf{fix}\ k\ \{x_i : t_i := f_i\}_{i=1}^{n}$
nv	::=	$v\ v \mid \mathsf{let}\ x : t = e\ \mathsf{in}\ e$
	|	$\mathsf{match}\ v\ \mathsf{with}\ \{C_i\ x_1 \ldots x_{ar(C_i)} \Rightarrow e_i\}_{i=1}^{n}$
e	::=	$v \mid nv$
d	::=	$c := v$
P	::=	$\epsilon \mid d.P$

Fig. 2 Syntax for A-normal form programs.

we want to monadify and their dependencies, together with information about *pure* functions, which should not be monadified, and *special* functions, which are mapped to monadic actions.

As said above, our algorithm attempts to introduce the smallest possible number of monadic wrappers, and to do that it infers the impure arity of functions. There is however a caveat here: in some cases, we cannot infer the exact impure arity. For instance, higher-order functions have no way to know whether their argument functions have effects or not. So we have to assume the worse (i.e., wrap the monad for each arrow). Yet the actual function we pass as argument may have a larger impure arity. This means that it is not sufficient only to add `bind` at application points, and `ret` when we want a result to be in the monad; we also need to introduce some extra wrapping on functions when we want to decrease their impure arity.

3.2.1 Syntax

In **Fig. 2**, we define the syntax of our programs (as a subset of Gallina). Our types do not include type abbreviations: we assume they are already expanded. More importantly, we only allow values as right hand sides of definitions d, and we require that expressions e be in A-normal form [6], i.e., we only allow application of values to values ($v\,v$), which makes sequencing of side-effects explicit. The transformation from the full syntax to A-normal forms is standard, and we give here a few of its rules (which are to be applied repeatedly until a normal form is reached).

$$\begin{array}{lll} v\,nv & \to & \mathsf{let}\ x : t = nv\ \mathsf{in}\ v\,x \\ nv\,e & \to & \mathsf{let}\ x : t = nv\ \mathsf{in}\ x\,e \\ \multicolumn{3}{l}{\mathsf{match}\ nv\ \mathsf{with}\ \{\ldots\}} \\ & \to & \mathsf{let}\ x : t = nv\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with}\ \{\ldots\} \end{array}$$

Here we assume that the type annotation t can be inferred from the context.

3.2.2 Transformation of Types

As said above, the *impure arity* is the number of arguments we have to apply before possibly exposing an effect. This number is similar to the type variable *strength* used for SML/NJ's weak type variables [7]. The impure arity is both an input and an output of the term transformation function $[\![e]\!]_\rho$, and is used as a guide when transforming types, in **Fig. 3**. If the impure arity is 0, then the corresponding term should be seen as a computation, and its type is wrapped in the monad. If the impure arity is greater than 0, then we can still view this term as pure, and we do not need to wrap its type. For inductive and function types, we need to apply

$$
\begin{array}{rcl}
[\![T]\!]_0 & = & M(T) \\
[\![T]\!]_1 & = & T \\
[\![I\ t_1 \dots t_n]\!]_0 & = & M\ (I\ [\![t_1]\!]_1 \dots [\![t_n]\!]_1) \\
[\![I\ t_1 \dots t_n]\!]_1 & = & I\ [\![t_1]\!]_1 \dots [\![t_n]\!]_1 \\
[\![t_1 \to t_2]\!]_0 & = & M\ ([\![t_1]\!]_1 \to [\![t_2]\!]_0) \\
[\![t_1 \to t_2]\!]_{i+1} & = & [\![t_1]\!]_1 \to [\![t_2]\!]_i
\end{array}
$$

Fig. 3 Transformation of types.

$$
\begin{array}{rcl}
[\![t]\!]_\rho & = & ([\![t]\!]_1, 1) \\
[\![x]\!]_\rho & = & (x, \rho(x)) \\
[\![c]\!]_\rho & = & (\nu(c), \rho(c)) \\
[\![C]\!]_\rho & = & (\hat{\nu}(C), \hat{\rho}(C)) \\
[\![\mathsf{fun}\ x : t \Rightarrow e]\!]_\rho & = & (\mathsf{fun}\ x : [\![t]\!]_1 \Rightarrow e', i+1) \\
\text{where } (e', i) & = & [\![e]\!]_{\rho[x \mapsto 1]} \\
[\![v_1\ v_2]\!]_\rho & = & (v_1'\ (v_2' \downarrow_1^j), i-1) \\
\text{where } (v_1', i) & = & [\![v_1]\!]_\rho \\
(v_2', j) & = & [\![v_2]\!]_\rho \\
\multicolumn{3}{l}{[\![\mathsf{let}\ x : t = e\ \mathsf{in}\ b]\!]_\rho} \\
& = & \left\{ \begin{array}{ll} (e' \gg\!= \mathsf{fun}\ x : [\![t]\!]_1 \Rightarrow b' \downarrow_0^i, 0) & \text{if } j = 0 \\ (\mathsf{let}\ x : [\![t]\!]_j = e'\ \mathsf{in}\ b', i) & \text{otherwise} \end{array} \right. \\
\text{where } (e', j) & = & [\![e]\!]_\rho \\
(b', i) & = & [\![b]\!]_{\rho[x \mapsto \max(1,j)]} \\
\multicolumn{3}{l}{[\![\mathsf{fix}\ k\ \{x_i : t_i := f_i\}_{i=1}^n]\!]_\rho} \\
& = & (\mathsf{fix}\ k\ \{x_i : [\![t_i]\!]_{j_i} := f_i' \downarrow_{j_i}^{j_i'}\}_{i=1}^n, j_k) \\
\text{where } j_i & = & FA(f_i) \\
\rho' & = & \rho[x_1 \mapsto j_1, \dots, x_n \mapsto j_n] \\
(f_i', j_i') & = & [\![f_i]\!]_{\rho'} \qquad (1 \le i \le n) \\
\multicolumn{3}{l}{[\![\mathsf{match}\ v\ \mathsf{with}\ \{C_i\ x_1 \dots x_{ar(C_i)} \Rightarrow e_i\}_{i=1}^n]\!]_\rho} \\
& = & (\mathsf{match}\ v'\ \mathsf{with}\ \{C_i\ x_1 \dots x_{ar(C_i)} \Rightarrow e_i' \downarrow_j^{j_i}\}_{i=1}^n, j) \\
\text{where } (v', j') & = & [\![v]\!]_\rho \\
(e_i', j_i) & = & [\![e_i]\!]_{\rho[x_1 \mapsto 1, \dots, x_{ar(C_i)} \mapsto 1]} \ (1 \le i \le n) \\
j & = & \min(j_1, \dots, j_n)
\end{array}
$$

Fig. 4 Transformation and impure arity for terms.

this transformation recursively (independently of whether the impure arity is 0 or not). On the left hand side of the arrow, or for parameters of inductive types, we do not know the real impure arity of the argument, so we must assume the worst; in a call-by-value language this means an impure arity of 1 (values themselves do not cause effects). On the right hand side of the arrow, since we are considering the term after one application, we have to decrease the impure arity by 1, or keep 0 if the term was already impure (contrary to full fledged effect systems, which annotate effects on each arrow, we only track the first effect, and assume that all subsequent arrows may be effectful). Note that we do not transform inductive types definitions, and as a result this transformation does not support having impure functions as arguments of constructors inside inductive definitions. Transforming inductive type definitions is not difficult in itself, but the resulting code would be rather inefficient, and this would make it difficult to reuse existing proofs on such data structures, so we just avoid it. On the other hand, it is fine to use function types as parameters, i.e., return a pair of functions for instance (pair is an inductive type in Coq).

3.2.3 Transformation of Terms

In **Fig. 4**, we define the transformation on terms $[\![e]\!]_\rho$, which should of course match the intended type. It takes as input a term e and two bindings: ν from constant or constructor names to constant names, and ρ from variables and constant or constructor names to their impure arities (at least 1); and returns a pair of a term and its impure arity. ν does not change during the transformation of a term, so we omit it in the notation. These bindings

$$
\begin{array}{rcll}
e \downarrow_j^i & = & \mathsf{error} & \text{if } i < j \\
e \downarrow_i^i & = & e & \\
e \downarrow_0^1 & = & \mathsf{return}\ e & \\
(\mathsf{fun}\ x : t \Rightarrow b) \downarrow_0^{i+1} & = & \mathsf{return}\ (\mathsf{fun}\ x : t \Rightarrow b \downarrow_0^i) & \\
(\mathsf{fun}\ x : t \Rightarrow b) \downarrow_{j+1}^{i+1} & = & \mathsf{fun}\ x : t \Rightarrow b \downarrow_j^i & \\
e \downarrow_j^{i+2} & = & \mathsf{let}\ y : t \to t' = e\ \mathsf{in} & \\
& & (\mathsf{fun}\ x : t \Rightarrow y\, x) \downarrow_j^{i+2} & \\
& & \text{when } \Gamma \vdash e : t \to t' & \\
\\
FA(\mathsf{fun}\ x : t \Rightarrow e) & = & 1 + FA(e) & \\
FA(\mathsf{fix}\ k\ \{v_i : t_i := f_i\}) & = & FA(f_k) & \\
FA(e) & = & 0 & \text{otherwise} \\
\\
pure(\nu, e) & = & \forall c \in e,\ \nu(c) = c\ \wedge & \\
& & \forall C \in e,\ \hat{\nu}(C) = C\ \wedge & \\
\multicolumn{4}{r}{\Gamma \vdash e : t_1 \to \dots \to t_n \to t_0 \text{ where}} \\
\multicolumn{4}{r}{t_0, \dots t_n \text{ do not contain functions}} \\
\\
\hat{\nu}(C) & = & \left\{ \begin{array}{ll} \nu(C) & \text{if defined} \\ C & \text{otherwise} \end{array} \right. & \\
\hat{\rho}(C) & = & \left\{ \begin{array}{ll} \rho(C) & \text{if defined} \\ ar(C) + 1 & \text{otherwise} \end{array} \right. &
\end{array}
$$

Fig. 5 Auxiliary functions for the transformation.

are initialized according to the parameters of monadification. I.e., to associate an action to a *special* constant or constructor, one sets $\nu(c)$ to the action and $\rho(c)$ to its arity. Interestingly, this can also be used to have the transformation ignore some functions, which are deemed *pure*, by setting $\nu(c) = c$ and $\rho(c) = ar(c) + 1$. $ar(c)$ denotes the arity of c, which is inferred from its type. This means that no monad will be inserted for these functions, and that their definition will be used unchanged. Note that, to keep the transformation coherent, higher-order functions cannot be declared (or inferred) to be pure; but they may be called unchanged inside functions declared pure. By convention, we also require that in other cases, the name of the action be different from the constant it is bound to, i.e., $\nu(c) = c$ only for pure functions or constructors.

The transformation uses five auxiliary functions (see **Fig. 5**). $\downarrow_j^i$ lowers the impure arity from i to j, wrapping the monad M wherever the impure arity is lowered to 0. The order of the clauses is relevant: $e \downarrow_j^{i+2}$ will only be eta-expanded if e is not of the form ($\mathsf{fun}\ x : t \Rightarrow b$), otherwise the previous clauses shall be used. *FA* counts the number of outside abstractions of a term, as a rough approximation of its impure arity, to be used only when transforming recursive functions. *pure* decides whether a definition should be handled as pure or not (i.e., there is no need to monadify it). This requires that it only contains pure constants and constructors, and that its type should be first-order. $\hat{\nu}$ and $\hat{\rho}$ extend ν and ρ for constructors, stating that constructors are pure by default.

Let us get back to the transformation. On types, we use the transformation for types, and return 1 as impure arity (no side effect). On variables, constants, and constructors, the transformation does nothing, and returns just their assumed impure arity. For a constructor C, this would usually be $ar(C) + 1$, i.e., the arity of the constructor plus 1, assuming no side-effect, but we may choose to replace a constructor by an action to make some observation, such as counting the number of times a constructor was used. In any case, since values cannot have immediate side-effects, the impure arity should be at least 1.

$$
\begin{array}{rcl}
[\![\epsilon]\!]_{\nu,\rho} & = & \epsilon \\
[\![c := v.\ P]\!]_{\nu,\rho} & = & \\
\multicolumn{3}{l}{\begin{cases} [\![P]\!]_{\nu,\rho} & \text{if } \nu(c) \text{ defined} \\ [\![P]\!]_{\nu[c\mapsto c],\rho[c\mapsto ar(c)+1]} & \text{if } \mathit{pure}(\nu, v) \\ c' := v'.\ [\![P]\!]_{\nu[c\mapsto c'],\rho[c\mapsto i]} & \text{otherwise} \end{cases}} \\
\text{where } (v', i) & = & [\![v]\!]_{\rho} \\
\text{and } c' & = & \text{fresh name}
\end{array}
$$

Fig. 6 Transformation of programs.

For functions, we assume that the argument has an impure arity of 1, and increase the impure arity of the result by 1. Conversely for applications, we decrease the impure arity of the function by 1 (since it is a value, its impure arity must be at least 1).

For let-expressions, we have 2 cases, depending on the impure arity of the bound expression. If it is 0, then it may have side-effects, so we insert a monadic bind, and lower the impure arity of the body to 0. If it is not less than 1, then there is no immediate effect, and the impure arity is that of the body, assuming the inferred arity for the bound expression.

The remaining cases just propagate the impure arity. For fix, we need to make a conservative assumption for recursive calls, using *FA*. For match, we lower all the branches to the lowest inferred impure arity.

3.2.4 Transformation of Programs

Finally, in **Fig. 6**, $[\![P]\!]_{\nu,\rho}$ takes a program P, and generates new bindings for parts of P which need to be transformed. These new bindings come in addition to the existing ones, which are kept unchanged. When transforming a binding $c := v$, one first has to decide whether a new binding is required. This can be avoided either if $\nu(c)$ is defined, meaning that c is declared as pure or as a monadic action, or if v is inferred pure, which requires extending ν and ρ to declare it as pure in the remainder of the program. Otherwise, a fresh name is generated, and a new transformed binding is added, extending ν and ρ to have the remainder of the program use this new binding.

3.2.5 Correctness of the Transformation

We have not formally proved that the transformation preserves typing. However, the result of the transformation goes through Coq's type checker, so if typing were broken, this would be discovered at that point. We have not formally proved that the intended call-by-value semantics is preserved either. However, we can easily check that it is the case in absence of side-effects, by proving that e and $[\![e]\!]_\rho$ are equal for the identity monad.

3.3 Implementation of Monadification

In this section, we explain the commands that our plugin provides to control the algorithm explained in the previous section (Section 3.2).

3.3.1 Monad Registration

Our plugin registers a monad by means of the following commands:

```
Monadify Type type_constructor
Monadify Return return_function
Monadify Bind bind_function
```

`Monadify Type`, `Monadify Return` and `Monadify Bind` registers the monadic triple. For example, the option monad already seen in Section 3.1 is registered as follows:

```
Monadify Type option.
Monadify Return @ret.
Monadify Bind @bind.
```

3.3.2 Monadification Traversal

The following command triggers monadification:

```
Monadification function/constructor ...
```

`Monadification x ...` monadifies a function or a constructor `x`. It also monadifies functions and constructors referenced from `x`. Concretely, it traverses functions/constructors from the function/-constructor specified as an argument. The traversal is depth-first order and may define monadified function in post-order.

The naming convention for monadified definitions is as follows. The name of the monadified definition is the concatenation of the original name without module names and "`M`". For example, `mathcomp.ssreflect.ssrnat.addn` becomes `addnM`. If the name is already in use, apostrophes are appended until an unused name is found.

3.3.3 Control of Monadification

To control monadification, it is possible to declare beforehand some functions to have or not side-effects. For each function or constructor traversed, if it is declared with `Monadify Action`, it is considered to have side-effect. If it is declared with `Monadify Pure`, it is considered to have no side-effect. `Monadify Pure` cannot take a higher order function because higher order functions may have effects when given an effectful function as argument. In the absence of a specific declaration, constructors are considered to have no side-effect and functions are examined as follows. Higher-order functions or functions referencing a function or constructor which has a side-effect, are considered to have a side-effect and a monadified definition is created. Otherwise they are considered to be pure.

We now explain the precise syntax and semantics of commands controlling monadification:

```
Monadify Action function/constructor ⇒ action
Monadify Pure function ...
```

`Monadify Action x ⇒ y` declares that `x` has a side-effect. When `x` is a n-arguments function or constructor of type $t_1 \to \ldots \to t_n \to t$, `y` should be a function with the type $t_1 \to \ldots \to t_n \to M\ t$. `x` is replaced with `y` during the monadification. This prevents the monadification of `x` (i.e., monadification does not transform the definition of `x`).

`Monadify Pure x ...` specifies that the monadified definition should use the function `x` directly without monadification. This prevents the monadification of `x` even if it contains actions.

The last command implemented by our plugin is `Monadify Reset`. It removes the information registered by `Monadify Type`, `Monadify Return`, `Monadify Bind`, `Monadify Action`, `Monadify Pure`.

3.4 Examples of Application of Monadification

3.4.1 Integer Overflow

The following function computes the kth power of `a`:

```
Fixpoint pow a k :=
 match k with
 | 0 ⇒ 1
 | S k' ⇒ a ∗ pow a k'
 end.
```

Our goal is to ensure the absence of integer overflow when natural numbers (of type `nat`) are replaced by finite-size integers. For that purpose, we introduce monadic versions of arithmetic functions that perform overflow tests:

```
(* check overflow on muln and S. *)
Definition check x :=
 if Nat.log2 x < 32 then Some x else None.
Definition mulM a b := check (a ∗ b).
Definition SM a := check a.+1.
```

We now monadify the `pow` function using the monad defined above (and the commands explained in Sections 3.3.2 and 3.3.3):

```
Monadify Action muln ⇒ mulM.
Monadify Action S ⇒ SM.
Monadification pow.
```

The result of the monadification of `pow` is as follows:

```
Fixpoint powM (a k : nat) :=
 match k with
 | 0 ⇒ SM 0
 | S k' ⇒ powM a k' >>= mulM a
 end.
```

As a result, we can make clear under which conditions the original function `pow` and its monadified version `powM` are semantically equivalent:

```
Theorem powM_ok :
 ∀ a b, Nat.log2 (pow a b) < 32 →
 (powM a b) = Some (pow a b).
Proof. ... Qed.
```

In other words, we have established that we can safely replace natural numbers with 32-bit integers as long as the precondition of the above theorem is met.

3.4.2 Buffer Overrun

The following example is a monadified function that fails when the `nth` function (that perform access to list elements given their indices) is used with an out-of-bounds index:

```
Definition nthM {T} (x0 : T) s n :=
 if n < size s then Some (nth x0 s n) else None.
Monadify Action nth ⇒ @nthM.
```

We will develop more in details this example with a realistic application to buffer overrun detection in Section 6.5.1.

3.4.3 Complexity Check

There are two list-reversal functions in the Coq standard library: the naive definition `rev`

```
Fixpoint rev (l:list A) : list A :=
 match l with
  | [] ⇒ []
  | x :: l' ⇒ rev l' ++ [x]
 end.
```

and the tail-recursive definition `rev'` (in `Coq.Lists.List`)

```
Fixpoint rev_append (l l': list A) : list A :=
 match l with
  | [] ⇒ l'
  | a::l ⇒ rev_append l (a::l')
 end.
Definition rev' l : list A := rev_append l [].
```

Let us introduce the following monad in order to count the number of explicit `cons` invocations at the Coq level:

```
Definition counter_with A : Type := nat ∗ A.
Definition ret {A} (x : A) := (0, x).
Definition bind {A} {B}
  (x' : counter_with A) (f : A → counter_with B) :=
 let (m, x) := x' in
 let (n, y) := f x in
 (m+n, y).

Monadify Type counter_with.
Monadify Return @ret.
Monadify Bind @bind.

Definition consM {T} (hd : T) tl := (1, cons hd tl).
Monadify Action cons ⇒ @consM.
```

Using the above monad, we can prove that `rev` performs $\frac{n(n+1)}{2}$ (where n is the length of the input list) invocations of `cons` whereas `rev'` only performs n such calls:

```
Monadification rev rev'.
Lemma NumConsInRev T s : revM T s =
 (((length s) ∗ (length s).+1)./2, rev s).
Lemma NumConsInRev' T s : rev'M T s = (length s, rev' s).
```

This provides a formal evidence that the time complexity of `rev` is at least quadratic while `rev'` is at least linear.

4. Generation of C Code

The generation of C from Gallina is implemented in the monomorphization plugin. We explain how datatypes are handled in Section 4.2, how code is generated for functions in Section 4.3, and for expressions in Section 4.4. This will make clear which subset of Gallina is supported. Section 4.1 provides a practical overview of the plugin implementation, in particular the naming convention for generated functions. See Section 6.3 for an extensive example of code generation.

4.1 Plugin Input/Output and Naming Conventions

The generation of C code is performed by the following command:

`GenC` $function\_name_0$ $function\_name_1$ ...

Input functions are expected to be monomorphized (as explained in Section 2). The output (i.e., the source code of generated C functions) is displayed in the terminal for `coqtop`, or in the message window of CoqIDE.

The name of generated C functions is built using the name of the input Coq function and its arity. Consider a Coq module *m* and a function *m.c* of type $t_1 \to ... \to t_n \to t$. The name of the generated C function becomes the name of the Coq function prefixed with `n` and the number of arguments, i.e., `n`*nc* (the module name is stripped). For example, the generated C function name for the monomorphized, two-arguments function `Top._addn`[*5] is `n2_addn`. The arity is computed using the formal arguments of the function definition and the actual arguments of the function invocation[*6].

Besides the name of the generated C function, one also needs to care about the names of generated types and variables. Let us provide a bit more insight to help understand the C code generated in Sections 4.3, 4.4, and later.

- The generated C function contains local variables (in addition to function arguments). They correspond to variables bound by `let` and `fun`. Gallina allows for conflicting names because they are just for display: internally, Coq uses de Bruijn indices. In C, we generate unique variable names using a global counter *i*, so that the variable *N* becomes `v`*i*`_`*N* (see Section 6.3 for an example).
- Characters unusable in a C identifier (such as the apostrophe `'`) are substituted with an underscore.
- The generated C code contains types such as the return type of the function, which are generated according to the naming rules explained in Section 2.6.

4.2 Implementation of Inductive Types

The implementation of C datatypes is not generated automatically: it must be provided by the user. Of course, one expects the implementation of basic datatypes such as integers to be standard and indeed we can provide default implementations for them. But in general, there is a great flexibility in the choice of representation and the user often wants a specific implementation in order to take advantage of the C language execution performance. We claim that user-customizable implementation of datatypes is actually a must-have feature for C code generation.

Concretely, for each inductive type used by the Coq function, the user provides a C version of (1) its type, (2) a constructor, (3) field accessors, and (4) several macros for the `switch` statement[*7]. See Section 4.4.4 for explanations about how the field accessors and the macros for the `switch` statements are used.

Here follow examples of basic Gallina datatypes implemented in C (Section 6.4 provides more examples).

4.2.1 Implementation of Pairs of Booleans

The example shown in Section 2.1 uses the type `prod bool bool` which itself uses the inductive types `bool` and `prod` defined in Coq standard library (module `Coq.Init.Datatypes`):

```
Inductive bool : Set := true : bool | false : bool.
```

```
Inductive prod (A B : Type) : Type :=
 pair : A → B → prod A B.
```

In C, the `bool` type can be implemented using `bool` of `stdbool.h` as follows:

```
#include <stdbool.h> /* defines bool type */
#define n0_true() true
#define n0_false() false
#define sw_bool(b) (b)
#define case_true_bool default
#define case_false_bool case false
```

In C, `prod bool bool` can be implemented by using the lower 2 bits of `int` to represent the two `bool`s:

```
#define prod_bool_bool int
#define n2_pair_bool_bool(x, y) ((x) | ((y) << 1))
#define field0_pair_prod_bool_bool(v) ((v) & 1)
#define field1_pair_prod_bool_bool(v) (((v) & 2) >> 1)
```

4.2.2 Implementation of Natural Numbers

The Coq natural numbers `nat` are defined in the Coq standard library (module `Coq.Init.Datatypes`) as follows:

```
Inductive nat : Set := O : nat | S : nat → nat.
```

In C, we can implement `nat` using `uint64_t`:

```
#define nat uint64_t
```

The `O` constructor of natural number can be implemented by the integer `0` with a cast, and the successor constructor `S` can be implemented as an increment by one:

```
#define n0_O() ((nat)0)
#define n1_S(n) ((n)+1)
```

There is only one field accessor for successors of natural numbers, implemented as the predecessor function:

```
#define field0_S_nat(n) ((n)-1)
```

Last, one provides a function and labels for `switch` statements (we explain how they are used in Section 4.4.4):

```
#define sw_nat(n) (n)
#define case_O_nat case 0
#define case_S_nat default
```

Using the above implementation, we can now provide efficient functions to perform arithmetic operations on "natural numbers" using C operators:

```
#define n2_addn(a,b) ((a)+(b))
#define n2_subn(a,b) ((a)-(b))
#define n2_muln(a,b) ((a)*(b))
#define n2_divn(a,b) ((a)/(b))
#define n2_modn(a,b) ((a)%(b))
```

They are safe as long as the application does not use values greater than or equal to 2^{64} (which is provable using our monadification plugin, as explained in Section 3).

4.3 From Coq Functions to C Functions

An important feature of our generation scheme is that it sup-

[*5] More precisely, this is the monomorphized version of `mathcomp.ssreflect.ssrnat.addn`, which performs addition of natural numbers.

[*6] If the number of formal arguments and the number of actual arguments are different (e.g., in the case of partial application), an error occurs. We plan to handle partial application in future work (see Section 9.1 for details).

[*7] In the case of a polymorphic type, the user has to provide them for each monomorphic instance.

ports proper tail recursion (i.e., tail recursion without stack consumption). The current extraction facility of Coq fails to provide this feature (see Appendix A.1 for a concrete example). This partly comes from the fact that Gallina has no loop construct but recursion to process arbitrary-size data. In contrast, C enjoys `goto` statements, which we can take advantage of.

Let us explain how we generate C code for function definitions. For that purpose, we consider a Coq constant c, which is a function of type $t_1 \to ... \to t_n \to t$. The name of the generated C function is $\mathtt{n}nc$ (as explained in Section 4.1).

When c is not recursive, we generate the following definition:

```
t nnc(t1 x1, ..., tn xn)
{ /* body */ }
```

(The translation of the body is the matter of Section 4.4.)

Let us now assume that c is a non-mutually recursive function. In this case, we add a label for tail recursion in front of the function body:

```
t nnc(t1 x1, ..., tn xn)
{ nnc:; /* body */ }
```

The case of mutually recursive functions is more technical and explained in Appendix A.2.

The final case is when c is a constant definition belonging to an inductive type. It is considered as a zero-argument function. Concretely, `Definition` $c : t := e$. where t is an inductive type is translated to:

```
t n0c(void)
{ /* code for e */ }
```

4.4 Expression Translation

In this section, we explain how we generate C code for Gallina expressions [*8]: (local) variables, `let` expressions, (global) constants, function application, and `match` expressions.

Since we need a trustful code generation, we do not perform any optimization. Also, we do not support higher-order functions (such as `map`). In other words, we are not trying to implement a full-fledged ML-to-C compiler.

4.4.1 Variables and `let` Expressions

The translation of Coq variables and `let` expressions is direct. A (local) variable in Coq is translated to a local variable in C. For example, the `let` expression

`let` $x : t := e$ `in` b

is translated to a variable initialization:

```
t x = /* code for e */; /* code for b */
```

If one cannot translate e to a C expression (in the case of a `match` expression for example), a variable declaration and an assignment to it are generated:

```
t x; /* code for e to assign x */; /* code for b */
```

4.4.2 Constants

After the monomorphization, the type of a Coq constant should be a function type or an inductive type.

At the time of this writing, a function should appear at the function position of an application because we do not support closures yet. The C code generation for this kind of function application is described in Section 4.4.3.

For a constant definition using an inductive type, a zero-argument function call is generated, as explained in Section 4.3. This is because a constant definition is translated to a zero-argument function definition.

4.4.3 Function Applications

The basic idea is to translate a function application in Coq to a function call in C. In particular, this is the case when one finds at function position a Coq constant that is a function. This basic idea extends to recursive calls.

There is nevertheless a special case of uttermost importance for predictability of tail recursion. It is when one finds at function position a local variable referencing a recursive function *and* when this application is at tail position. In this case, we generate (1) assignments to arguments and (2) a `goto` statement.

4.4.4 `match` Expressions

A `match` expression is translated to a `switch` statement.

Let us consider the `match` expression `match` e_0 `with` $\{C_i\ x_1 \ldots x_{ar(C_i)} \Rightarrow e_i\}_{i=1}^{n}$, where the expression e_0 has an inductive type $t = I\ t_1\ \ldots\ t_m$. I is defined as follows:

`Inductive` $I\ (T_1 \ldots T_m :$ `Type`$) :=$
...
$|\ C_i : t_{i1} \to ... \to t_{iar(C_i)} \to I\ T_1\ \ldots\ T_m$
...

The `match` expression is translated to the following `switch` statement:

```
switch (sw_t(e0))
{
  /* branches of match, i = 1...n */
  ...
  case_Ci_t: {
    /* obtain the field values, j = 1...ar(Ci) */
    ...
    t'ij xij = field(j − 1)_Ci_t(e0);
    ...
    /* code for ei */
  }
  ...
}
```

The monomorphic type t'_{ij} is the polymorphic type t_{ij} in which $T_1 \ldots T_m$ have been instantiated with $t_1 \ldots t_m$.

The constructs `sw_`t, `case_`C_i`_`t, and `field`j`_`C_i`_`t have been provided by the user (see Section 4.2):

- `sw_`t is a function which returns an integer that identifies the constructor of e_0;
- `case_`C_i`_`t is a macro which expands to the `default` label or to a `case` label;
- `field`j`_`C_i`_`t is the field accessor to the jth argument of constructor C_i (j is a zero-origin index).

If I has only one constructor, e_1 is always chosen. Therefore, only

[*8] In Sections 4.2 and 4.3 we were dealing with inductive types and function declarations that are defined using the `Inductive` and `Definition` commands; technically, they are part of the Vernacular language of Coq, not Gallina.

the code to obtain field values and the code for e_1 (and no `switch` statement) are generated. In this case, `sw_`*t* and `case_`C_i`_`*t* are not used.

5. Validating Experiment: Monadification of SSReflect's `seq.v`

As a validating experiment, we tested the monadification plugin by attempting to monadify the 49 list-related functions defined in the `seq.v` file of SSReflect: `all`, `allpairs`, `behead`, `belast`, `cat`, `catrev`, `constant`, `count`, `drop`, `filter`, `find`, `flatten`, `foldl`, `foldr`, `has`, `head`, `incr_nth`, `index`, `iota`, `iter`, `last`, `map`, `mask`, `mkseq`, `ncons`, `nilp`, `nth`, `ohead`, `pairmap`, `perm_eq`, `pmap`, `rem`, `reshape`, `rev`, `rot`, `rotr`, `scanl`, `seqn`, `set_nth`, `shape`, `size`, `subseq`, `sumn`, `take`, `undup`, `uniq`, `unzip1`, `unzip2` and `zip`.

We registered `S` and `cons` as monadic actions, meaning that all functions constructing natural numbers or lists would require monadification.

Only 7 functions did not call (transitively) `S` or `cons`, and as a result were deemed pure and not monadified: `behead`, `drop`, `head`, `last`, `nth`, `ohead` and `subseq`.

Of the remaining 42 functions, 36 were successfully monadified, and proved semantically equal to the original by applying the identity monad.

6 functions could not be monadified, and had to be declared pure, together with 2 other internal definitions, to complete the monadification. Namely `constant`, `index`, `perm_eq`, `undup`, `uniq`, `nat_eqType` and `seq_eqType` use higher-order constructors, and `seqn` uses a dependent type. These are the 2 categories of types which our translation does not support.

`seq.v` itself is not representative of the code we expect users to write, but our goal here was rather to show that it is possible to translate code relying on existing libraries. From this point of view, the fact that a small number of functions cannot be used directly should not be a major problem.

6. A Realistic Application: the *rank* Algorithm

In this section, we illustrate our scheme for the generation of C code using a realistic example: the *rank* algorithm (Section 6.1). For the sake of clarity, our explanations focus on only one of the sub-routines. First, we display and comment on the result of monomorphization (Section 6.2) and of the generation of C code (Section 6.3). Second, we explain the C implementation of the data structures (Section 6.4). Last, we explain how to use monadification to (1) guarantee the absence of integer overflows and buffer overruns despite the change of data structures from natural numbers to finite-size integers (Section 6.5.1), and (2) prove the time complexity of the algorithm (Section 6.5.2).

6.1 Background: the *rank* Algorithm

The *rank* algorithm is an algorithm for succinct data structures (see Section 1.1). It can be implemented as a function that counts bits in a prefix of a bitstring.

rank can be defined naively in Gallina by counting the b (0 or 1) bits in the first i bits of the bitstring s:

```
Definition rank b i s := count_mem b (take i s).
```

The problem with such an implementation is that its time complexity is $O(i)$.

The true *rank* algorithm has time complexity $O(1)$, using preprocessed auxiliary data of size $o(n)$, which is generated in $O(n)$ time complexity (where n is the length of s). In Ref. [26], we implemented this algorithm using the function `rank_init` (already used as an illustrating example in Section 2.4) to perform preprocessing and the function `rank_lookup`. We also proved that these functions are correct in the sense that they implement the same operation as the naive `rank` function seen above:

```
Lemma RankCorrect b s i : i <= bsize s →
 rank_lookup (rank_init b s) i = rank b i s.
```

6.2 Monomorphization of the *rank* Function

Hereafter, we focus on one sub-routine of the *rank* algorithm, namely the `buildDir2` function. (The complete source code can be found online [27].) It is a function that deals with the construction of the auxiliary data referred to in Section 6.1:

```
Fixpoint buildDir2 b s sz2 c i D2 m2 :=
 if c is cp.+1 then
  let m := bcount b i sz2 s in
  buildDir2 b s sz2
   cp (i + sz2) (pushD D2 m2) (m2 + m)
 else
  (D2, m2).
```

We first monomorphize the `buildDir2` function to the `_buildDir2` function using the `Monomorphization` command (Section 2.2):

```
Fixpoint _buildDir2 b s sz2 c i D2 m2 :=
 match c with
 | 0 ⇒ _pair_DArr_nat D2 m2
 | cp.+1 ⇒
   let m := _bcount b i sz2 s in
   let n := _addn i sz2 in
   let d := _pushD D2 m2 in
   let n0 := _addn m2 m in
   _buildDir2 b s sz2 cp n d n0
 end.
```

We observe in particular that the polymorphic Gallina constructor invocation `(D2, m2)` is monomorphized to `_pair_DArr_nat D2 m2`.

6.3 C Code Generation for the *rank* Function

We now generate C code for the function `_buildDir2` obtained by monomorphization in Section 6.2. For that purpose, we use the `GenC` command (Section 4.1). It generates the following `n7_buildDir2` C function [*9]:

```
prod_DArr_nat
n7_buildDir2(bool v10_b, bits v9_s, nat v8_sz2,
```

[*9] As explained in Section 4.1, the prefix `n7` is generated because the function has 7 arguments.

```
  nat v7_c, nat v6_i, DArr v5_D2, nat v4_m2)
{
  n7_buildDir2:;
  switch (sw_nat(v7_c))
  {
    case_O_nat: {
      return n2_pair_DArr_nat(v5_D2, v4_m2);
    }
    case_S_nat: {
      nat v12_cp = field0_S_nat(v7_c);
      nat v13_m = n4_bcount(v10_b,v6_i,v8_sz2,
                            v9_s);
      nat v14_n = n2_addn(v6_i, v8_sz2);
      DArr v15_d = n2_pushD(v5_D2, v4_m2);
      nat v16_n = n2_addn(v4_m2, v13_m);
      v7_c = v12_cp;
      v6_i = v14_n;
      v5_D2 = v15_d;
      v4_m2 = v16_n;
      goto n7_buildDir2;
    }
  }
}
```

We observe that the Gallina function arguments `b`, `s`, etc., are translated to the C function arguments `v10_b`, `v9_s`, etc. Gallina variables such as `cp`, `m`, etc., are translated to C variables, `v12_cp`, `v13_m`, etc [*10]. The Gallina `if` expression (actually, syntactic sugar for a `match` expression) is translated to a `switch` statement. The Gallina function call `_pair_DArr_nat D2 m2` is translated to the C function invocation `n2_pair_DArr_nat(v5_D2, v4_m2)`. The Gallina non-tail function invocations, `bcount`, etc., are translated to the C function invocations `n4_bcount`, etc. In contrast, the Gallina tail recursion to `buildDir2` is translated to assignments to formal arguments (`v7_c = v12_cp`, etc.) and a `goto` statement (`goto n7_buildDir2`).

As already stated above, `buildDir2` is only one sub-routine of the *rank* algorithm. We also generate C functions for the Gallina functions `pred`, `neq0`, `buildDir1`, `buildDir`, `rank_init` and `rank_lookup`. Among these functions, `pred` (notation ".−1" in SSReflect) is defined in the Coq standard library. The complete script to generate the C source code of the *rank* function and the C source code in itself can be found online [27].

6.4 C Data Types and Primitives for the *rank* Function

6.4.1 C Data Types Provided by the User

As already explained in Section 4.2, we need to provide C implementations of Gallina inductive types for our generated C code to be runnable. Regarding the *rank* algorithm, we implement the C data types corresponding to the following inductive types: `bits`, `DArr`, `prod_DArr_DArr`, `prod_DArr_nat`, `prod_prod_DArr_DArr_nat`, and `Aux`. We also use the C implementation of `bool` and `nat` described in Sections 4.2.1 and 4.2.2.

[*10] The prefix v*i* makes variable names unique, as explained in Section 4.1.

Most of the types above (except `bits` and `DArr`, which are discussed below) are implemented with simple `struct`s whose fields correspond to the constructor arguments. This is possible because they have only one constructor. They are defined as a non-pointer type, so that no heap memory is used. For example, here follows the implementation of `prod_DArr_DArr`. Its constructor (`n2_pair_DArr_DArr`) is implemented as a compound literal:

```
typedef struct {
  DArr D1;
  DArr D2;
} prod_DArr_DArr;
#define n2_pair_DArr_DArr(D1, D2) \
  ((prod_DArr_DArr){ (D1), (D2) })
```

The field accessors are references to structure members:

```
#define field0_pair_prod_DArr_DArr(x) ((x).D1)
#define field1_pair_prod_DArr_DArr(x) ((x).D2)
```

The implementations of `bits` and `DArr` are more involved.

The Gallina type `bits` is a wrapper for `seq bool`, but we implement it as a bitstring because it is more memory-efficient, can be built in linear-time, and provide arbitrary element-lookup in constant-time. Concretely, `bits` is a C `struct` that contains a pointer to a `bits_heap` `struct` that is allocated in the heap to store the actual contents (see Ref. [26], Section 4.2 for details):

```
typedef struct {
  uint64_t *buf;
  nat len; /* current length [bit] */
  nat max; /* maximum length [bit] */
} bits_heap;

typedef struct {
  bits_heap *heap;
  nat len; /* expected length [bit] */
} bits;
```

We do not provide the constructor and field accessors for `bits` because they convert from/to the `seq bool` type, which is too memory-consuming. For fast processing, we provide the C functions `n1_bsize` and `n4_bcount`:

```
#define n1_bsize(s) ((s).len)
nat n4_bcount(bool b, nat n, nat m, bits bs)
```

The Gallina type `DArr` implements an array of small integers whose elements are less than 2^w. It is implemented using `bits`. We assume $0 < w$ (assumption to be justified using monadification in Section 6.5.1):

```
typedef struct {
  nat w;
  bits s;
} DArr;
DArr n1_emptyD(nat w) { ... }
DArr n2_pushD(DArr d, nat n) { ... }
nat n2_lookupD(DArr d, nat i) { ... }
#define n1_sizeD(d) (n1_bsize(d) / (d).w)
```

6.4.2 C Primitives Provided by the User

In addition to C data types, the user has the possibility to provide hand-written functions. It happens that, in the case of our *rank* function, the user actually *has* to do so because there are a few functions that rely on the euclidean division of SSReflect (this limitation was already mentioned in Section 2.3). This is the case for example of the `bitlen` function. Fortunately, it can be implemented much more efficiently (and, from a programmer's viewpoint, more naturally) using the gcc builtin `__builtin_clzl` as follows:

```
static inline nat
n1_bitlen(nat n)
{
  if (n == 0) return 0;
  assert(64 <= sizeof(long) * CHAR_BIT);
  return 64 - __builtin_clzl(n);
}
```

6.5 Monadification of the *rank* Function

6.5.1 *rank* Function Never Fails

In this section, we use our monadification plugin (Section 3) to show that our *rank* algorithm written in Coq (more precisely, the `rank_init` and `rank_lookup` functions discussed in Sections 6.1 and 6.2) does not suffer from integer overflow and buffer overrun problems (and therefore that the change of data structures explained in Section 6.4 is safe).

Of course, the absence of integer overflows and buffer overruns is conditional. More precisely, we prove that `rank_init` and `rank_lookup` never fail as long as the length of the input bitstring is less than 2^{64}. This is the best implementation (in terms of the number of inputs it can handle) that one can achieve using the `uint64_t` integral type (which is used as the return type of the *rank* function).

In order to check potential integer overflows for addition, multiplication and the successor function, we introduce the following monadic actions:

```
Definition W := 64.
Definition check x := if log2 x < W then Some x
  else None.
Definition addM a b := check (a + b).
Definition mulM a b := check (a * b).
Definition SM a := check a.+1.
Monadify Action addn ⇒ addM.
Monadify Action muln ⇒ mulM.
Monadify Action S ⇒ SM.
```

We also check for divisions by zero with the following monadic actions:

```
Definition divM a b := if b is 0 then None
  else Some (a %/ b).
Definition modM a b := if b is 0 then None
  else Some (a %% b).
Monadify Action divn ⇒ divM.
Monadify Action modn ⇒ modM.
```

Regarding buffer overrun, we identify a potential issue with the `bcount` function. `bcount b n m s` counts the number of *b* (0 or 1) bits within the *m* bits following the index *n* into the bitstring *s*. The following monadic version ensures the absence of buffer overruns (see line 2) [*11]:

```
1 Definition bcountM (b : bool) n m (s : bits) :=
2   if n + m <= bsize s then
3     check (bcount b n m s)
4   else
5     None.
6 Monadify Action bcount ⇒ bcountM.
```

There is also a number of requirements to be met for the `DArr` C data structure seen in Section 6.4.

As described in Section 6.4, our C-level implementation of `DArr` assumes that $0 < w$. We check this assumption in the function `emptyD`:

```
Definition emptyDM w := if w is 0 then None
  else Some (emptyD w).
Monadify Action emptyD ⇒ emptyDM.
```

The array of small integers, `DArr`, uses only *w* bits for a single element. The element must therefore be less than 2^w. We check this condition in the function `pushD`:

```
Definition pushDM D v := let: darr w d := D in
  if v < 2 ^ w then Some (pushD D v) else None.
Monadify Action pushD ⇒ pushDM.
```

When performing lookups within `Darr`, the index must be less than the length of the array. We check this condition in `lookupD`:

```
Definition lookupDM D i :=
  if i < sizeD D then check (lookupD D i) else None.
Monadify Action lookupD ⇒ lookupDM.
```

Now that all the above monadic actions have been registered, we can use the `Monadification` command to generate monadic versions of `rank_init` and `rank_lookup`, and then prove formally that the monadic versions behave as the original versions:

```
Lemma RankSuccess b s i :
  let n := bsize s in log2 n < W → i <= n →
  (rank_initM b s >>= fun aux ⇒ rank_lookupM aux i) =
  Some (rank_lookup (rank_init b s) i).
```

The lemma `RankSuccess` is a formal evidence that once the Gallina code of the *rank* algorithm is turned into C, the data structures introduced in Section 6.4 are safe, and that the C code will not fail because of integer overflows, buffer overruns, etc.

6.5.2 The Complexity of *rank* Function

In this section, we use the monadification plugin to establish the complexity of our *rank* function.

We compute the number of bits examined in `rank_init` and `rank_lookup`. Since `rank_init` and `rank_lookup` use `bcount` to examine bits in the input bitstring, we use the monad described in Section 3.4.3 with the following action:

```
Definition bcountM (b : bool) n m (s : bits) :=
```

[*11] It also checks for the absence of integer overflows at line 3.

```
  (m, bcount b n m s).
Monadify Action bcount ⇒ bcountM.
```

The following lemma shows that the number of bits examined by `rank_init` is $O(n)$:

```
Lemma RankInitNumBitsExamined b s :
  let n := bsize s in rank_initM b s =
  (n %/ (bitlen n).+1 ∗ (bitlen n).+1, rank_init b s).
```

The following lemma shows that the number of bits examined by `rank_lookup` is $O(1)$ when `uint64_t` is used as described in Section 6.5.1:

```
Lemma RankLookupNumBitsExamined b s i :
  let aux := rank_init b s in
  let n := bsize s in rank_lookupM aux i =
  (i %% (bitlen n).+1, rank_lookup aux i).
```

It is because the maximum of `i %% (bitlen n).+1` is 64 when $\log_2 n < 64$. (Usually it is implemented using a table or an instruction such as POPCNT on recent Intel CPUs.)

This is the expected time complexity of the *rank* function for the succinct data structures [*12].

7. Evaluation of the Trusted Base of Our C Code Generation Scheme

It turns out that our C code generation scheme only modestly increases Coq's trusted base.

The main piece we add to Coq's trusted base is the C code generator (Section 4). It is part of the monomorphization plugin that also contains the implementation of monomorphization (Section 2). This plugin consists of the OCaml source files shown in **Table 1**. Among these files, we do not count `monomorph.ml` as part of the trusted base because the result of monomorphization can be verified inside Coq (as explained in Section 2.1). That is why we claim that the monomorphization plugin adds less than 1,000 lines of code to the trusted base.

This is small in comparison to the standard Coq extraction OCaml code. This is of course no surprise, since we only target a subset of Gallina (see Section 2.5). For reference, **Table 2** shows the OCaml source files that constitute the Coq standard extraction (`plugins/extraction` directory in Coq 8.6). We can see that the extraction to OCaml code adds more than 6,000 lines to the trusted base (this does not include `haskell.ml`, `scheme.ml`, `json.ml`, and `big.ml`).

The implementation of C data structures should be regarded as part of the trusted base. In our generation scheme, they must be provided by the user (see Section 4.2) but this is not different from standard extraction to OCaml code where there is often a need to replace automatically generated OCaml data structures with more efficient ones.

This comparison does not take into account the monadification plugin (Section 3) because its main role is to guarantee the safe use of fixed-size integer types (e.g., `uint64_t` in C and `int` in OCaml) in lieu of natural numbers (integer overflows, array accesses, etc.). Its use is relevant to both our C code generation scheme and the standard Coq extraction.

Table 1 Number of lines of the C code generator.

filename	lines	contents
`g_monomorph.ml4`	30	register commands
`monoutil.ml`	136	utilities
`monomorph.ml`	710	monomorphization
`genc.ml`	696	C code generator

8. Related Work

8.1 Generation of C code from Gallina via JSON

coq2c [20] is a tool written in Haskell to convert Gallina programs written in a monadic style to C. It uses as an intermediate representation the JSON output generated by the standard Coq extraction facility. The main purpose of coq2c is the Pip [21] protokernel.

coq2c targets Gallina programs written in an imperative style using a monad, whereas our plugins target programs in purely functional style. coq2c itself is not proved, so that it comes as an addition to the trusted base. In our case, only the C code generator (without monomorphization) is added to the trusted base; it is therefore smaller than the Coq standard extraction facility extended with coq2c.

8.2 Gallina to C Compiler Written in Gallina

The CertiCoq project [2] has been developing a compiler from Gallina to C. It does not use the standard Coq extraction facility. The compiler is written in Gallina and proved in Coq. It provides a way to run Gallina programs in a certified environment using the CompCert C compiler [14].

CertiCoq tries to generate optimized code, that therefore does not look like hand-written C code. Developers are considering modifying CompCert to support a different calling convention in order to support proper tail calls, whereas our C generation scheme supports only proper tail recursion as a design decision.

8.3 Standard Coq Extraction

There exist several extensions of the Coq extraction facility that support various languages: Scala [*13] [9], SML [*14], Erlang [*15],

Table 2 Number of lines of the standard Coq extraction.

filename	lines	contents
`g_extraction.ml4`	152	register commands
`common.ml`	648	utilities
`extract_env.ml`	682	extraction commands
`extraction.ml`	1098	from Coq terms to MiniML
`mlutil.ml`	1524	utilities over ML types
`modutil.ml`	411	functions upon ML modules
`table.ml`	921	parameters for extraction
`ocaml.ml`	773	OCaml code generator
`haskell.ml`	398	Haskell code generator
`scheme.ml`	236	Scheme code generator
`json.ml`	274	JSON generator
`big.ml`	154	utilities for `ExtrOcamlZBigInt.v`

[*12] The time of operations other than `bcount` is proportional to `bcount`'s at most. `bcount` is invoked at the innermost recursion (function `buildDir2`) in `rank_init`. `rank_lookup` has no recursion.

[*13] https://github.com/hemmi/coq2scala for Coq 8.4pl2
[*14] https://github.com/fetburner/Coq2SML for Coq 8.4pl4
[*15] https://github.com/tcarstens/verlang for Coq 8.4pl2

Rust [16], Ruby [17], Python [18].

They do not seem to be maintained, since they have not been updated to the current version of Coq (version 8.6). It is difficult to use them with a newer Coq version without compiling Coq because they are distributed in the form of patches or as a modified Coq distribution. In comparison, our extraction mechanism is easier to test with a newer Coq version because it is implemented as plugins that can be used without installing Coq itself.

8.4 Monomorphization

Many languages use monomorphization, which produces specialized copies of code: Ada generics [17], C++ template [24], D template [1], Pizza's heterogeneous translation [18], MLton [4], [30], and Rust [3]. Monomorphization is common for system programming languages. It is efficient and obeys C++'s zero-overhead principle [25]. Monomorphization can cause exponential code bloat in the worst case, but it is not a problem in practice.

Our monomorphization is similar to MLton's, which is no surprise since Gallina is similar to ML (see Section 2.5 for more insights).

8.5 Monadification

Previous work about monadification are inadequate for our purpose because they introduce too many or too few monads compared to our needs (see Section 3.1). There are also technical reasons for which they cannot be used directly in Coq.

8.5.1 Hatcliff and Danvy's Monadification

Hatcliff and Danvy described monadification algorithms for several evaluation strategies including call-by-value [8]. Their algorithm encapsulates all values into the monad, including functions, which are completely currified. For example, the two-arguments function $f : t_1 \to t_2 \to t_3$ is transformed to $f' : M\ (t_1 \to M\ (t_2 \to M\ t_3))$, and the application $f\ a\ b$ is transformed to

$$f' \gg= \lambda g.\ a' \gg= \lambda x.\ g\ x \gg= \lambda h.\ b' \gg= \lambda y.\ h\ y$$

where $a' : M\ t_1$ and $b' : M\ t_2$ are monadified versions of a and b.

We have already explained in Section 3.1 why the systematic insertion of the type constructor M is not satisfactory. In addition, although Hatcliff and Danvy's transformation works well for the simply typed lambda-calculus, it cannot be used as such for Gallina. Indeed, Coq may reject the result because (1) the detection of the decreasing argument of structural recursion fails when it is not the first argument, and (2) type-checking may fail when type arguments are passed via monadic operations. Appendix A.3 provides concrete examples of these technical problems.

Our monadification algorithm tries to preserve the original definitions as much as possible, and, as a consequence, retains type arguments and decreasing arguments as regular arguments. Problem (2) above could also be solved by applying after monadification the monad law (return x) $\gg= f \equiv f\ x$ but this provides only a partial solution.

[16] https://github.com/pirapira/coq2rust for Coq 8.4
[17] https://github.com/mzp/coq-ruby for Coq 8.4
[18] http://coqcots.gforge.inria.fr/coq8.4pl3-python/ for Coq 8.4pl3

8.5.2 Erwig's Monadification

Erwig proposed a monadification algorithm that performs selective introduction of monads [5]. It tries to insert much less monads than Hatcliff and Danvy's algorithm by transforming

$$f : t_1 \to \ldots \to t_k \to t_{k+1} \to \ldots \to t_n$$

to

$$f' : t_1 \to \ldots \to t_k \to M\ (t_{k+1} \to \ldots \to t_n).$$

As explained in Section 3.1, these are not the positions at which we expect monads to appear because some effects may still happen after further applications. Moreover, the monad used needs to be *runnable*, i.e., it should have an operation "$\mathsf{run} : M\ t \to t$", which allows to discard the monad, possibly causing a runtime exception if there is no value to return. This is not suitable for Coq, which has no exception mechanism, and would defeat our goal of using monads to prove properties of programs.

The modification algorithm we introduced in Section 3 performs a selective introduction of monads, but is not as aggressive as Erwig's.

8.6 C Code Generation

There exist several compilers that produce C code as an output but they all stumble on difficulties inherent to C:

- A C compiler cannot implement proper tail call with usual calling conventions [22].
- C has no special mechanism to detect integer overflow.
- It is difficult to implement closures and garbage collection in C.

Our C code generator generates proper tail recursion (not proper tail call) using `goto`, the absence of dynamic integer overflow can be ensured by proofs using monadification, and a conservative garbage collector such as the Boehm GC can be used. We do not support closures yet.

Bigloo [23] is a Scheme compiler which generates C code. Pre-Scheme [11] is a statically typed Scheme subset for the Scheme48 [10] virtual machine implementation. They implement proper tail recursion using `goto` similarly to us. They check integer overflow dynamically. Bigloo uses Boehm GC. Pre-Scheme has no GC (because it is used to implement Scheme48's GC).

sml2c [29] is a Standard ML compiler that generates C code based on SML/NJ. It uses the UUO driver (trampoline) for proper tail call. As a consequence, function calls have an overhead comparable with native function calls in C. Also, function arguments are passed via global variables. The generated C code is therefore very different from the ML source code. Integer overflow is checked dynamically. sml2c uses the GC of SML/NJ.

8.7 Native Code Generation

Recently, LLVM [12] is becoming popular as a compiler framework providing portable code generation. This is a new approach to native code generation. However, using an LLVM backend makes it difficult to obtain information (function signatures and macros) in C header files. They are important to use features provided by the compiler and the operating system. Also, LLVM is not usable if some project requires a (non-LLVM) C compiler supported by a vendor.

9. Conclusion

In this paper, we proposed a new scheme for code generation that outputs provably-safe C code for the Coq proof-assistant. Concretely, we introduced one plugin for monomorphization (Section 2) and C code generation (Section 4). Monomorphization transforms in a provable way polymorphic functions written in Gallina to monomorphic functions, so as to facilitate C code generation. Thanks to this pre-processing, the translation to C code becomes direct. It provides efficient code, preserving in particular tail recursion and using optimized data structures to implement Coq inductive types. If the C data structure and the Coq inductive type behave differently (such as C's `int` and Coq's `nat`), one can prove that the difference does not cause any problem using our monadification plugin (Section 3). We demonstrated the usability of our monadification plugin with the `seq.v` library of SSReflect (Section 5). As a realistic use-case, we used the *rank* algorithm of succinct data structures (Section 6) for which we generated C code using customized data structures (`uint64_t` for Coq's `nat` and a memory-efficient bitstring data structure). Thanks to our approach, we were able to prove not only functional correctness, but also to ensure formally the good behavior of C data structures, as well as the time complexity of the *rank* algorithm. We argued that our approach relies on a small trusted base (Section 7), and therefore provides a trustful and pragmatic generation scheme for verified, realistic low-level C code.

9.1 Future Work

As short-term future work, we plan a number of technical improvements: provide default implementations for standard inductive types (so as to reduce the user's burden), address the issue of name conflicts that the naming rules of Section 2.6 may cause (at the time of this writing, one candidate solution is to use name mangling like in C++).

As a next step, we plan to support a pluggable GC and closures. It is important to be able to choose between various GCs in the event that the generated code is linked to, say, OCaml or Ruby (in which case, the GC of the latter should be used). Closures would make it possible to support partial application. Concretely, we would like to be able to define functions with less arguments than the formal ones and functions that return closures. The issues of GC and closures are related because both mechanisms need to be compatible.

Regarding use-cases, we plan to experiment our scheme for C code generation with other algorithms for succinct data structures, such as the *select* function or wavelet trees.

As mid-term future work, we will investigate extension of monomorphization to other arguments than type arguments so as to open the door to the generation of even more efficient code. For example, it would make it possible to specialize the *rank* function w.r.t. one of its arguments (say, `sz2` instantiated with 64); small blocks would then be accessible in an aligned manner so as to use the `POPCNT` instruction of recent Intel CPUs. We will also investigate support for destructive update. For that purpose, we are considering implementing a Coq plugin to perform linear typing. Using this plugin, it would be possible to detect, for C code generated in pure functional style, when destructive update can be performed safely.

Acknowledgments This work is supported by JSPS KAKENHI Grant Number 15K12013.

References

[1] Alexandrescu, A.: *The D programming language*, Addison-Wesley Professional (2010).

[2] Anand, A., Appel, A.W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger, O.S., Sozeau, M. and Weaver, M.: CertiCoq: A verified compiler for Coq, *The Third International Workshop on Coq for Programming Languages* (2017).

[3] Anderson, B., Bergstrom, L., Goregaokar, M., Matthews, J., McAllister, K., Moffitt, J. and Sapin, S.: Engineering the servo web browser engine using Rust, *38th International Conf. Software Engineering Companion* (*ICSE 2016*), pp.81–89 (2016).

[4] Cejtin, H., Jagannathan, S. and Weeks, S.: Flow-directed closure conversion for typed languages, *European Symp. Programming* (*ESOP 2000*), LNCS, Vol.1782, pp.56–71 (2000).

[5] Erwig, M. and Ren, D.: Monadification of functional programs, *Science of Computer Programming*, Vol.52, No.1–3, pp.101–129 (2004).

[6] Flanagan, C., Sabry, A., Duba, B.F. and Felleisen, M.: The Essence of Compiling with Continuations, *ACM SIGPLAN Conf. Programming Language Design and Implementation* (*PLDI 1993*), pp.243–247 (1993).

[7] Greiner, J.: Weak polymorphism can be sound, *Journal of Functional Programming*, Vol.6, No.1, pp.111–141 (1996).

[8] Hatcliff, J. and Danvy, O.: A generic account of continuation-passing styles, 21st ACM *Symp. Principles of Programming Languages* (*POPL 1994*), pp.458–471 (1994).

[9] Hemmi, K., Tanabe, Y., Imai, Y. and Hagiya, M.: Verified Code Extraction from Coq to Scala (in Japanese), *31st JSSST Annual Meeting of the Japan Society for Software Science and Technology* (*JSSST 2014*) (2014).

[10] Kelsey, R.A. and Rees, J.A.: A tractable Scheme implementation, *Lisp and Symbolic Computation*, Vol.7, No.4, pp.315–335 (1994).

[11] Kelsey, R.A.: Pre-Scheme: A Scheme Dialect for Systems Programming (online), available from ⟨http://mumble.net/~kelsey/papers/prescheme.ps.gz⟩ (accessed 2017-11-07).

[12] Lattner, C. and Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation, *Proc. International Symposium on Code Generation and Optimization: Feedback-directed and runtime optimization*, IEEE Computer Society (2004).

[13] Letouzey, P.: Certified functional programming: Program extraction within Coq proof assistant, PhD thesis, Université Paris-Sud, France (2004).

[14] Leroy, X.: Formal verification of a realistic compiler, *Comm. ACM*, Vol.52, No.7, pp.107–115 (2009).

[15] MLton: Monomorphise (online), available from ⟨http://mlton.org/Monomorphise⟩ (accessed 2017-05-02).

[16] Moggi, E.: Notions of computation and monads, *Information and Computation*, Vol.93, No.1, pp.55–92 (1991).

[17] Morrison, R., Dearle, A., Connor, R.C.H. and Brown, A.L.: An ad hoc approach to the implementation of polymorphism, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.3, pp.342–371 (1991).

[18] Odersky, M. and Wadler, P.: Pizza into Java: Translating theory into practice, *24th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* (*POPL 1997*), Paris, pp.146–159 (1997).

[19] Okanohara, D.: *The world of fast character string analysis* (In Japanese), Iwanami Shoten (2012).

[20] Oudjail, V. and Hym, S.: coq2c (online), available from ⟨https://github.com/2xs/coq2c⟩ (accessed 2017-05-02).

[21] The Pip Development Team: The Pip Protokernel (online), available from ⟨http://pip.univ-lille1.fr/⟩ (accessed 2017-05-02).

[22] Probst, M.: *Proper tail recursion in C*, Diplomarbeit, Institute of Computer Languages, Vienna University of Technology (2001).

[23] Serrano, M. and Weis, P.: Bigloo: A portable and optimizing compiler for strict functional languages, *International Symp. Static Analysis* (*SAS 1995*), LNCS, Vol.983, pp.366–381 (1995).

[24] Stroustrup, B.: *The C++ Programming Language*, Third Edition, Addison-Wesley (1997).

[25] Stroustrup, B.: Abstraction and the C++ machine model, *International Conference on Embedded Software and Systems*, LNCS, Vol.3605, pp.1–13, Springer Berlin Heidelberg (2004).

[26] Tanaka, A., Affeldt, R. and Garrigue, J.: Formal Verification of the rank Algorithm for Succinct Data Structures, *18th International Conference on Formal Engineering Methods* (*ICFEM 2016*), pp.242–260

(2016).
[27] Tanaka, A., Affeldt, R. and Garrigue, J.: Formal Verification of the rank Algorithm for Succinct Data Structures, project website, available from ⟨https://staff.aist.go.jp/tanaka-akira/succinct/⟩ (accessed 2017-07-12). Contains references to plugins' implementation and Coq scripts for this article.
[28] Tanaka, A.: Coq Bug 4312 - ocamlopt can not optimize tail call in a extracted function (online), available from ⟨https://coq.inria.fr/bugs/show_bug.cgi?id=4312⟩ (accessed 2017-05-02).
[29] Tarditi, D., Lee, P. and Acharya, A.: No assembly required: Compiling standard ML to C, *ACM Letters on Programming Languages and Systems*, Vol.1, No.2, pp.161–177 (1992).
[30] Tolmach, A.P. and Oliva, D.: From ML to Ada: Strongly-typed language interoperability via source translation, *Journal of Functional Programming*, Vol.8, No.4, pp.367–412 (1998).

Appendix

A.1 Extraction Robustness w.r.t. Tail Recursion

This section provides a concrete illustration for the issue Robustness.

Extraction is not robust and too sensitive to customization. In particular, when one customizes the match construct, this leads to the introduction of anonymous functions and tail recursion may not be preserved by extraction and leads to stack consumption [*19].

For example, the following Coq function extracted and compiled with ocamlopt (the native compiler of OCaml) consumes stack [28]:

```
Fixpoint f (i a1 a2 a3 a4 a5 a6 a7 a8 a9 : nat) :=
 match i with
 | O ⇒ O
 | S j ⇒ f j a1 a2 a3 a4 a5 a6 a7 a8 a9
 end.
Require Import ExtrOcamlNatInt.
Extraction f.
```

Here is the result of extraction to OCaml:

```
let rec f i a1 a2 a3 a4 a5 a6 a7 a8 a9 =
 (fun f0 fS n → if n=0 then f0 () else fS (n−1))
  (fun _ → 0)
  (fun j → f j a1 a2 a3 a4 a5 a6 a7 a8 a9)
  i
```

The extracted code consists of mutually recursive functions: `f` and two anonymous functions `(fun f0 fS → ...)` (from the module `ExtrOcamlNatInt` of the Coq standard library) and `(fun j → ...)`, which corresponds to a branch of the original match construct. Although all recursive calls are tail call, on the AMD64 architecture, ocamlopt fails to compile them so as to avoid stack consumption.

A.2 Generation of C Code for Mutually Recursive Coq Functions

Mutually recursive functions in Coq are defined as follows:

Fixpoint $c_1\ (x_{11} : t_{11}) \ldots (x_{1n_1} : t_{1n_1}) : t_1 := \ldots$
...
with $c_i\ (x_{i1} : t_{i1}) \ldots (x_{in_i} : t_{in_i}) : t_i := \ldots$
...
with $c_m\ (x_{m1} : t_{m1}) \ldots (x_{mn_m} : t_{mn_m}) : t_m := \ldots$

We generate C code in such a way that tail recursion between functions is translated to gotos to avoid stack consumption. A C function *bodyfunc* is generated that contains all the functions' bodies. This function takes as arguments (1) one argument to choose a function body and (2) actual arguments for the chosen function. These actual arguments are passed via a void* pointer to a struct. Similarly, the return value of each body is passed via a void* pointer. We generate entry functions $\mathtt{n}n_ic_i$, which build the struct for the arguments and the return buffer, and invoke *bodyfunc*.

More precisely, the generated code is as follows (*1–4*):

1. Declaration of structs The structs for arguments are declared:

```
/* struct definitions for arguments, i = 1...m */
... struct c_i { t_i1 x_i1; ...; t_in_i x_in_i; }; ...
```

2. Forward Declaration of bodyfunc A forward declaration is necessary because the entry functions invoke *bodyfunc*. It takes an int to choose a function, a pointer to the arguments, and a pointer to the buffer for the return value:

```
void bodyfunc(int i, void *argsp, void *retp);
```

3. Generation of Entry Functions They store the arguments to a struct and invoke *bodyfunc* with the int that identifies the appropriate body (zero-origin). The return value should be stored in `ret`:

```
/* entry functions, i = 1...m */
...
t_i nn_ic_i(t_i1 x_i1, ..., t_in_i x_in_i) {
  struct c_i args = { x_i1, ..., x_in_i };
  t_i ret;
  bodyfunc(i − 1, &args, &ret);
  return ret;
}
...
```

4. Generation of bodyfunc It contains at the top declarations of arguments which can be assignable by any function involved in tail recursion. A switch statement is used for dispatching to functions' bodies. In each case, the arguments are retrieved from the struct and a label is declared for tail recursion. The function body is translated in such a way that the result is assigned to the return value buffer provided by the entry function. After that, the control-flow returns to the entry function:

```
/* body function */
void bodyfunc(int i, void *argsp, void *retp)
{ /* argument declarations, i = 1...m, j = 1...n_i */
  ... t_ij x_ij; ...
  /* dispatch to the specified function body */
  switch (i) {
   default: /* suppress warning */
```

*19 We observe that this happens with native compilation when the number of arguments is greater than the number of registers. The bytecode generator does not seem to suffer from this defect.

```
    /* function bodies, i = 1...m */
    ...
    case i-1: {
      /* argument retrieval, j = 1...n_i */
      ... x_ij = ((struct c_i *)argsp)->x_ij; ...
      nn_ic_i:; /* label for tail recursion */
      /* body to assign (*(t_i*)retp)) */
      return;
    }
    ...
  }
}
```

There is a number of opportunities for simplifications in the above generic transformation scheme:

- struct for arguments can be eliminated if the arguments are the same for all functions.
- The return value buffer can be eliminated if the return type is the same for all functions.
- The label for tail recursion can be eliminated if the function is not tail-called.
- If the function is always tail-called, the entry function and the case label can be eliminated.
- If only one function can be called as a non-tail call, the entry function and *bodyfunc* can be merged to one function and the switch statement can be eliminated.

A.3 Hatcliff and Danvy's Monadification Applied to Gallina

This section provides concrete evidence that Hatcliff and Danvy's monadification algorithm cannot be used directly in Coq. We have observed in particular failures to identify the decreasing argument and type-checking issues (as mentioned in Section 8.5.1).

Example of Failed Detection of the Decreasing Argument The following function is a Gallina implementation of the "power function":

```
(* direct style: *)
Fixpoint pow a k := match k with
            | O ⇒ 1
            | S k' ⇒ a * pow a k'
          end.
```

Coq infers that it terminates by detecting that k is its decreasing argument. Once the power function has been monadified, the detection of the decreasing argument fails because it is not passed directly to the f binder of the fix operator for recursive functions:

```
(* monadic style: *)
Definition powM := return fix f a := return fun k ⇒
 match k with
 | O ⇒ return 1
 | S k' ⇒ f a >>= fun g ⇒ g k' >>= fun b ⇒ return a * b
 end.
(* Error: Recursive definition of f is ill-formed.
   Recursive call to f has principal argument
   equal to "a" instead of a subterm of "a". *)
```

Example of Type-check Failure for a Monadified Function The example below shows that the monadification of the polymorphic identity function fails to type-check when applied to the type of natural numbers because Coq does not infer that the type argument T is in fact nat:

```
(* direct style: *)
Definition id T (x : T) := x.
Check id nat 1.
(* monadic style: *)
Definition idM := return fun T ⇒
 return fun (x : T) ⇒ return x.
Check idM >>= fun f ⇒
   (return nat) >>= fun T ⇒
   f T >>= fun g ⇒
   (return 1) >>= fun x ⇒
   g x.
(* Error: The term "x" has type "nat" while it is
   expected to have type "T". *)
```

Akira Tanaka was born in 1972. He received his Ph.D. from JAIST in 2000. His research interest is programming languages. He is currently a Senior Researcher at AIST.

Reynald Affeldt received his M.S. and Ph.D. degrees in computer science from the University of Tokyo in 2001 and 2004 respectively. He was a Research Scientist in the Graduate School of Information Science and Technology of the University of Tokyo, and joined AIST in 2005 as a Research Scientist specializing in software verification applied to secure computing. He is now a member of the Information Technology Research Institute.

Jacques Garrigue received his M.S. degree from University Paris 7, and D.S. degree from the University of Tokyo in 1995. He is alumnus of École Normale Supérieure in Paris. He was Research Associate at Kyoto University from 1995 to 2004, and has been Associate Professor at Nagoya University since 2004. His interests are in the theory of programming languages, particularly type systems and proof of programs. He is a member of IPSJ, JSSST and ACM.