Scallina: Translating Verified Programs from Coq to Scala

Youssef El Bakouny(✉) and Dani Mezher

CIMTI, ESIB, Saint-Joseph University, Beirut, Lebanon
{Youssef.Bakouny,Dany.Mezher}@usj.edu.lb

Abstract. This paper presents the Scallina prototype: a new tool which allows the translation of verified Coq programs to Scala. A typical workflow features a user implementing a functional program in Gallina, the core language of Coq, proving this program's correctness with regards to its specification and making use of Scallina to synthesize readable Scala components.

This synthesis of readable, debuggable and traceable Scala components facilitates their integration into larger Scala or Java applications; opening the door for a wider community of programmers to benefit from the Coq proof assistant. Furthermore, the current implementation of the Scallina translator, along with its underlying formalization of the Scallina grammar and corresponding translation strategy, paves the way for an optimal support of the Scala programming language in Coq's native extraction mechanism.

Keywords: Formal methods · Functional programming · Compiler
Coq · Scala

1 Introduction

In our modern world, software bugs are becoming increasingly detrimental to the engineering industry. As a result, we have recently witnessed interesting initiatives that use formal methods, potentially as a complement to software testing, with the goal of proving a program's correctness with regards to its specification. A remarkable example of such an initiative is a U.S. National Science Foundation (NSF) expedition in computing project called "the Science of Deep Specification (DeepSpec)" [17].

Since the manual checking of realistic program proofs is impractical or, to say the least, time-consuming; several proof assistants have been developed to provide machine-checked proofs. Coq [12] and Isabelle/HOL [14] are currently two of the world's leading proof assistants; they enable users to implement a program, prove its correctness with regards to its specification and extract a proven-correct implementation expressed in a given functional programming language. Coq has been successfully used to implement CompCert, the world's first formally verified C compiler [8]; whereas Isabelle/HOL has been successfully

S. Ryu (Ed.): APLAS 2018, LNCS 11275, pp. 131–145, 2018.
https://doi.org/10.1007/978-3-030-02768-1_7

used to implement seL4, the world's first formally verified general-purpose operating system kernel [7]. The languages that are currently supported by Coq's extraction mechanism are OCaml, Haskell and Scheme [11], while the ones that are currently supported by Isabelle/HOL's extraction mechanism are OCaml, Haskell, SML and Scala [4].

The Scala programming language [15] is considerably adopted in the industry. It is the implementation language of many important frameworks, including Apache Spark, Kafka, and Akka. It also provides the core infrastructure for sites such as Twitter, Coursera and Tumblr. A distinguishing feature of this language is its practical fusion of the functional and object-oriented programming paradigms. Its type system is, in fact, formalized by the calculus of Dependent Object Types (DOT) which is largely based on path-dependent types [1]; a limited form of dependent types where types can depend on variables, but not on general terms.

The Coq proof assistant, on the other hand, is based on the calculus of inductive constructions; a Pure Type System (PTS) which provides fully dependent types, i.e. types depending on general terms [3]. This means that Gallina, the core language of Coq, allows the implementation of programs that are not typable in conventional programming languages. A notable difference with these languages is that Gallina does not exhibit any syntactic distinction between terms and types [12].[1]

To cope with the challenge of extracting programs written in Gallina to languages based on the Hindley-Milner [5,13] type system such as OCaml and Haskell, Coq's native extraction mechanism implements a theoretical function that identifies and collapses Gallina's logical parts and types; producing untyped λ-terms with inductive constructions that are then translated to the designated target ML-like language, i.e. OCaml or Haskell. During this process, unsafe type casts are inserted where ML type errors are identified [10]. For example, these unsafe type casts are currently inserted when extracting Gallina records with path-dependent types. However, as mentioned in Sect. 3.2 of [11], this specific case can be improved by exploring advanced typing aspects of the target languages. Indeed, if Scala were a target language for Coq's extraction mechanism, a type-safe extraction of such examples could be done by an appropriate use of Scala's path-dependent types.

It is precisely this Scala code extraction feature for Coq that constitutes the primary aim of the Scallina project. Given the advances in both the Scala programming language and the Coq proof assistant, such a feature would prove both interesting and beneficial for both communities. The purpose of this tool demonstration paper is to present the Scallina prototype: a new tool which allows the translation of verified Coq programs to Scala. A typical workflow features a user implementing a functional program in Coq, proving this program's correctness with regards to its specification and making use of Scallina to synthesize readable Scala components which can then be integrated into larger Scala or Java applications. In fact, since Scala is also interoperable with Java, such a feature

[1] Except that types cannot start by an abstraction or a constructor.

would open the door for a significantly larger community of programmers to benefit from the Coq proof assistant.

Section 2 of this paper exposes the overall functionality of the tool while Sect. 3 portrays its strengths and weaknesses and Sect. 4 concludes. The source code of Scallina's implementation is available online[2] along with a command line interface, its corresponding documentation and several usage examples.

2 Integrating Verified Components into Larger Applications

Coq's native extraction mechanism tries to produce readable code; keeping in mind that confidence in programs also comes via the readability of their sources, as demonstrated by the Open Source community. Therefore, Coq's extraction sticks, as much as possible, to a straightforward translation and emphasizes the production of readable interfaces with the goal of facilitating the integration of the extracted code into larger developments [9]. This objective of seamless integration into larger applications is also shared by Scallina. In fact, the main goal of Scallina is to extract, from Coq, Scala components that can easily be integrated into existing Scala or Java applications.

Although these Scala components are synthesized from verified Coq code, they can evidently not guarantee the correctness of the larger Scala or Java application. Nevertheless, the appropriate integration of such verified components significantly increases the quality-level of the whole application with regards to its correctness; while, at the same time, reducing the need for heavy testing.

Indeed, even if a purely functional Scala component is verified with regards to its specification, errors caused by the rest of the application can still manifest themselves in the code of this proven-correct component. This is especially true when it comes to the implementation of verified APIs that expose public higher-order functions. Take the case of Listing 1 which portrays a Gallina implementation of a higher-order `map` function on a binary tree Algebraic Data Type (ADT). A lemma which was verified on this function is given in Listing 2; whereas the corresponding Scala code, synthesized by Scallina, is exhibited in Listing 3.

Listing 1. A Gallina higher-order map function on a binary tree ADT

```
Inductive Tree A := Leaf | Node (v: A) (l r: Tree A).
Arguments Leaf {A}.
Arguments Node {A} _ _ _.
Fixpoint map {A B} (t: Tree A) (f: A → B) : Tree B :=
match t with
  Leaf => Leaf
| Node v l r => Node (f v) (map l f) (map r f)
end.
```

[2] https://github.com/JBakouny/Scallina/tree/v0.5.0.

Listing 2. A verified lemma on the higher-order map function

```
Definition compose {A B C} (g : B → C) (f : A → B) := fun x : A => g (f x).
Lemma commute : ∀ {A B C} (t: Tree A) (f: A → B) (g: B → C),
map t (compose g f) = map (map t f) g.
```

Listing 3. The synthesized Scala higher-order map function with the binary tree ADT

```
sealed abstract class Tree[+A]
case object Leaf extends Tree[Nothing]
case class Node[A](v: A, l: Tree[A], r: Tree[A]) extends Tree[A]
object Node {
  def apply[A] =
    (v: A) => (l: Tree[A]) => (r: Tree[A]) => new Node(v, l, r)
}
def map[A, B](t: Tree[A])(f: A => B): Tree[B] =
  t match {
    case Leaf         => Leaf
    case Node(v, l, r) => Node(f(v))(map(l)(f))(map(r)(f))
  }
```

Unlike Gallina, the Scala programming language supports imperative constructs. So, for example, if a user of the `map` function mistakenly passes a buggy imperative function `f` as second argument, the overall application would potentially fail. In such a case, the failure or exception would *appear* to be emitted by the verified component, even though the bug was caused by the function `f` that is passed as second argument, not by the verified component.

To fix such failures, most industrial programmers would first resort to debugging; searching for and understanding the root cause of the failure. Hence, the generation of Scala components that are both readable and debuggable would pave the way for a smoother integration of such formal methods in industry. The synthesized Scala code should also be traceable back to the source Gallina code representing its formal specification in order to clarify and facilitate potential adaptations of this specification to the needs of the overall application.

Therefore, in congruence with Coq's native extraction mechanism, the Scallina translator adopts a relatively straightforward translation. It aims to generate, as much as possible, idiomatic Scala code that is readable, debuggable and traceable; facilitating its integration into larger Scala and Java applications. We hope that this would open the door for Scala and Java programmers to benefit from the Coq proof assistant.

3 Translating a Subset of Gallina to Readable Scala Code

As mentioned in Sect. 1, Gallina is based on the calculus of inductive constructions and, therefore, allows the implementation of programs that are not typable in conventional programming languages. Coq's native extraction mechanism tackles this challenge by implementing a theoretical function that identifies and collapses Gallina's logical parts and types; producing untyped λ-terms with inductive constructions that are then translated to the designated target

ML-like language; namely OCaml or Haskell. During this translation process, a type-checking phase approximates Coq types into ML ones, inserting unsafe type casts where ML type errors are identified [11]. However, Scala's type system, which is based on DOT, significantly differs from that of OCaml and Haskell. For instance, Scala sacrifices Hindley-Milner type inference for a richer type system with remarkable support for subtyping and path-dependent types [1]. So, on the one hand, Scala's type system requires the generation of significantly more type information but, on the other hand, can type-check some constructs that are not typable in OCaml and Haskell.

As previously mentioned, the objective of the Scallina project is not to repeat the extraction process for Scala but to extend the current Coq native extraction mechanism with readable Scala code generation. For this purpose, it defines the Scallina grammar which delimits the subset of Gallina that is translatable to readable and traceable Scala code. This subset is based on an ML-like fragment that includes both inductive types and a polymorphism similar to the one found in Hindley-Milner type systems. This fragment was then augmented by introducing the support of Gallina records, which correspond to first-class modules. In this extended fragment, the support of Gallina dependent types is limited to path-dependent types; which is sufficient to encode system F [1].

The Scallina prototype then implements, for Gallina programs conforming to this grammar, an optimized translation strategy aiming to produce idiomatic Scala code similar to what a Scala programmer would usually write. For example, as exhibited by Listings 1 and 3, ADTs are emulated by Scala case classes. This conforms with Scala best practices [16] and is already adopted by both Isabelle/HOL and Leon [6]. However, note that Scallina optimizes the translation of ADTs by generating a `case object` instead of a `case class` where appropriate; as demonstrated by `Leaf`. Note also that this optimization makes good use of Scala's variance annotations and `Nothing` bottom type. This use of an object instead of a parameterless class improves both the readability and the performance of the output Scala code. Indeed, the use of Scala singleton object definitions removes the performance overhead of instantiating the same parameterless class multiple times. Furthermore, when compared to the use of a parameterless case class, the use of a case object increases the readability of the code by avoiding the unnecessary insertions of empty parenthesis. This optimization, embodied by our translation strategy, is a best practice implemented by Scala standard library data structures such as `List[+A]` and `Option[+A]`.

Since the identification and removal of logical parts and fully dependent types are already treated by Coq's theoretical extraction function, the Scallina prototype avoids a re-implementation of this algorithm but focuses on the optimized translation of the specified Gallina subset to Scala. This supposes that a prior removal of logical parts and fully dependent types was already done by Coq's theoretical extraction function and subsequent type-checking phase; catering for a future integration of the Scallina translation strategy into Coq's native extraction mechanism. In this context, Scallina proposes some modifications to the latter with regards to the typing of records with path-dependent types. These

modifications were explicitly formulated as possible future works through the `aMonoid` example in [11]. Listing 4 shows a slight modification of the `aMonoid` example which essentially removes its logical parts. While, as explained in [11], the current extraction of this example produces unsafe type casts in both OCaml and Haskell; Scallina manages to translate this example to the well-typed Scala code shown in Listing 5.

Listing 4. The aMonoid Gallina record with its logical parts removed

```
Record aMonoid : Type := newMonoid {
  dom : Type;
  zero : dom;
  op : dom → dom → dom
}.
Definition natMonoid := newMonoid nat 0 (fun (a: nat) (b: nat) => a + b).
```

Listing 5. The Scala translation of the aMonoid Gallina record

```
trait aMonoid {
  type dom
  def zero: dom
  def op: dom => dom => dom
}
def newMonoid[dom](zero: dom)(op: dom => dom => dom): aMonoid = {
  type aMonoid_dom = dom
  def aMonoid_zero = zero
  def aMonoid_op = op
  new aMonoid {
    type dom = aMonoid_dom
    def zero: dom = aMonoid_zero
    def op: dom => dom => dom = aMonoid_op
  }
}
def natMonoid = newMonoid[Nat](0)((a: Nat) => (b: Nat) => a + b)
```

Indeed, Scallina translates Gallina records to Scala functional object-oriented code which supports path-dependent types. In accordance with their Scala representation given in [1], record definitions are translated to Scala traits and record instances are translated to Scala objects. When a Gallina record definition explicitly specifies a constructor name, Scallina generates the equivalent Scala object constructor that can be used to create instances of this record, as shown in Listing 5; otherwise, the generation of the Scala record constructor is intentionally omitted. In both cases, Gallina record instances can be created using the named fields syntax {| ... |}, whose translation to Scala produces conventional object definitions or, where necessary, anonymous class instantiations. A complete and well-commented example of a significant Gallina record translation to conventional Scala object definitions is available online[3]. This example also contains a

[3] https://github.com/JBakouny/Scallina/tree/v0.5.0/packaged-examples/v0.5.0/list-queue.

proof showing the equivalent behavior, with regards to a given program, of two Scala objects implementing the same trait.

A wide variety of usage examples, available online[4], illustrate the range of Gallina programs that are translatable by Scallina. These examples are part of more than 325 test cases conducted on the current Scallina prototype and persisted by more than 7300 lines of test code complementing 2350 lines of program source code. A significant portion of the aforementioned Scallina usage examples were taken from Andrew W. Appel's Verified Functional Algorithms (VFA) e-book [2] and then adapted according to Scallina's coding conventions.

4 Conclusion and Perspectives

In conclusion, the Scallina project enables the translation of a significant subset of Gallina to readable, debuggable and traceable Scala code. The Scallina grammar, which formalizes this subset, facilitates the reasoning about the fragment of Gallina that is translatable to conventional programming languages such as Scala. The project then defines an optimized Scala translation strategy for programs conforming to the aforementioned grammar. The main contribution of this translation strategy is its mapping of Gallina records to Scala; leveraging the path-dependent types of this new target output language. Furthermore, it also leverages Scala's variance annotations and `Nothing` bottom type to optimize the translation of ADTs. The Scallina prototype shows how these contributions can be successfully transferred into a working tool. It also allows the practical Coq-based synthesis of Scala components that can be integrated into larger applications; opening the door for Scala and Java programmers to benefit from the Coq proof assistant.

Future versions of Scallina are expected to be integrated into Coq's extraction mechanism by re-using the expertise acquired through the development of the current Scallina prototype. In this context, an experimental patch for the Coq extraction mechanism[5] was implemented in 2012 but has since become incompatible with the latest version of Coq's source code. The implementation of Scallina's translation strategy into Coq's extraction mechanism could potentially benefit from this existing patch; updating it with regards to the current state of the source code. During this process, the external implementation of the Scallina prototype, which relies on Gallina's stable syntax independently from Coq's source code, could be used to guide the aforementioned integration; providing samples of generated Scala code as needed.

Acknowledgements. The authors would like to thank the National Council for Scientific Research in Lebanon (CNRS-L) (http://www.cnrs.edu.lb/) for their funding, as well as Murex S.A.S (https://www.murex.com/) for providing financial support.

[4] https://github.com/JBakouny/Scallina/tree/v0.5.0/src/test/resources in addition to https://github.com/JBakouny/Scallina/tree/v0.5.0/packaged-examples/.

[5] http://proofcafe.org/wiki/en/Coq2Scala.

A Appendix: Demonstration of the Scallina Translator

Scallina's functionalities will be demonstrated through the extraction of Scala programs from source Gallina programs. The fully executable version of the code listings exhibited in this demo are available online[6]. This includes, for both of the exhibited examples: the source Gallina code, the lemmas verifying its correctness and the synthesized Scala code.

A.1 Selection Sort

The selection sort example in Listing 6 is taken from the VFA e-book. It essentially portrays the translation of a verified program that combines `Fixpoint`, `Definition`, `let in` definitions, if expressions, pattern matches and tuples.

The source code of the initial program has been modified in accordance with Scallina's coding conventions. The exact changes operated on the code are detailed in its online version[7] under the `Selection.v` file.

Listing 6. The VFA selection sort example

```
Require Import Coq.Arith.Arith.
Require Import Coq.Lists.List.
Fixpoint select (x: nat) (l: list nat) : nat * (list nat) :=
match l with
|  nil => (x, nil)
|  h:: t => if x <=? h
               then let (j, l1) := select x t in (j, h:: l1)
               else let (j,l1) := select h t in (j, x:: l1)
end.
Fixpoint selsort (l : list nat) (n : nat) {struct n} : list nat :=
match l, n with
| x:: r, S n1 => let (y,r1) := select x r
               in y :: selsort r1 n1
| nil, _ => nil
| _:: _, 0 => nil
end.
Definition selection_sort (l : list nat) : list nat := selsort l (length l).
```

Listing 7 portrays the theorems developed in the VFA e-book which verify that this is a sorting algorithm. These theorems along with their proofs still hold on the example depicted in Listing 6.

[6] https://github.com/JBakouny/Scallina/tree/v0.5.0/packaged-examples/v0.5.0.
[7] https://github.com/JBakouny/Scallina/tree/v0.5.0/packaged-examples/v0.5.0/selection-sort.

Listing 7. The theorems verifying that selection sort is a sorting algorithm

```
(** Specification of correctness of a sorting algorithm:
it rearranges the elements into a list that is totally ordered. *)
Inductive sorted: list nat → Prop :=
 | sorted_nil: sorted nil
 | sorted_1: ∀ i, sorted (i:: nil)
 | sorted_cons: ∀ i j l, i <= j → sorted (j::l) → sorted (i::j:: l).
Definition is_a_sorting_algorithm (f: list nat → list nat) :=
  ∀ al, Permutation al (f al) ∧ sorted (f al).
Definition selection_sort_correct : Prop :=
  is_a_sorting_algorithm selection_sort.
Theorem selection_sort_perm:
  ∀ l, Permutation l (selection_sort l).
Theorem select_smallest:
  ∀ x al y bl, select x al = (y, bl) →
     Forall (fun z => y <= z) bl.
Theorem selection_sort_sorted: ∀ al, sorted (selection_sort al).
Theorem selection_sort_is_correct: selection_sort_correct.
```

The verified Gallina code in Listing 6 was translated to Scala using Scallina. The resulting Scala code is exhibited in Listing 8.

Listing 8. The synthesized Scala selection sort algorithm

```
import scala.of.coq.lang._
import Nat._
import Pairs._
import MoreLists._
object Selection {
  def select(x: Nat)(l: List[Nat]): (Nat, List[Nat]) =
    l match {
      case Nil => (x, Nil)
      case h :: t => if (x <= h) {
        val (j, l1) = select(x)(t)
        (j, h :: l1)
      }
      else {
        val (j, l1) = select(h)(t)
        (j, x :: l1)
      }
    }
  def selsort(l: List[Nat])(n: Nat): List[Nat] =
    (l, n) match {
      case (x :: r, S(n1)) => {
        val (y, r1) = select(x)(r)
        y :: selsort(r1)(n1)
      }
      case (Nil, _) => Nil
      case (_ :: _, Zero) => Nil
    }
  def selection_sort(l: List[Nat]): List[Nat] = selsort(l)(length(l))
}
```

A.2 List Queue Parametricity

The list queue example in Listing 9 is taken from the test suite of Coq's Parametricity Plugin[8]. It essentially portrays the translation of Gallina record definitions and instantiations to object-oriented Scala code. It also illustrates the use of Coq's Parametricity plugin to prove the equivalence between the behavior of several instantiations of the same record definition; these are then translated to object implementations of the same Scala trait.

The source code of the initial program has been modified in accordance with Scallina's coding conventions. The exact changes operated on the code are detailed in its online version[9] under the `ListQueueParam.v` file.

Listing 9. The parametricity plugin ListQueue example

```
Require Import List.
Record Queue := {
  t : Type;
  empty : t;
  push : nat → t → t;
  pop : t → option (nat * t)
}.
Definition ListQueue : Queue := {|
  t := list nat;
  empty := nil;
  push := fun x l => x :: l;
  pop := fun l =>
    match rev l with
      | nil => None
      | hd :: tl => Some (hd, rev tl) end
|}.
Definition DListQueue : Queue := {|
  t := (list nat) * (list nat);
  empty := (nil, nil);
  push := fun x l =>
    let (back, front) := l in
    (x :: back,front);
  pop := fun l =>
    let (back, front) := l in
    match front with
      | nil =>
        match rev back with
          | nil => None
          | hd :: tl => Some (hd, (nil, tl))
        end
      | hd :: tl => Some (hd, (back, tl))
    end
|}.
```

[8] https://github.com/parametricity-coq/paramcoq.

[9] https://github.com/JBakouny/Scallina/tree/v0.5.0/packaged-examples/v0.5.0/list-queue.

```
(* A non-dependently typed version of nat_rect. *)
Fixpoint loop {P : Type}
  (op : nat → P → P) (n : nat) (x : P) : P :=
  match n with
  | 0 => x
  | S n0 => op n0 (loop op n0 x)
  end.
(*
This method pops two elements from the queue q and
then pushes their sum back into the queue.
*)
Definition sumElems(Q : Queue)(q: option Q.( t)) : option Q.( t) :=
match q with
| Some q1 =>
  match (Q.( pop) q1) with
  | Some (x, q2) =>
    match (Q.( pop) q2) with
    | Some (y, q3) => Some (Q.( push) (x + y) q3)
    | None => None
    end
  | None => None
  end
| None => None
end.
(*
This programs creates a queue of n+1 consecutive numbers (from 0 to n)
and then returns the sum of all the elements of this queue.
*)
Definition program (Q : Queue) (n : nat) : option nat :=
(* q := 0::1::2::...::n *)
let q :=
  loop Q.( push) (S n) Q.( empty)
in
let q0 :=
  loop
  (fun _ (q0: option Q.( t)) => sumElems Q q0)
  n
  (Some q)
in
match q0 with
| Some q1 =>
  match (Q.( pop) q1) with
  | Some (x, q2) => Some x
  | None => None
  end
| None => None
end.
```

Listing 10 portrays the lemmas verifying the equivalence between the behavior of either `ListQueue` or `DListQueue` when used with the given program. The

proofs of these lemmas, which were implemented using Coq's Parametricity plugin, still hold on the example depicted in Listing 9. Instructions on how to install the Parametricity plugin to run these machine-checkable proofs are provided online.

Listing 10. The lemmas verifying the ListQueue parametricity example

```
Lemma nat_R_equal : ∀ x y, nat_R x y → x = y.
Lemma equal_nat_R : ∀ x y, x = y → nat_R x y.
Lemma option_nat_R_equal : ∀ x y, option_R nat nat nat_R x y → x = y.
Lemma equal_option_nat_R : ∀ x y, x = y → option_R nat nat nat_R x y.
Notation Bisimilar := Queue_R.
Definition R (l1 : list nat) (l2 : list nat * list nat) :=
 let (back, front) := l2 in
  l1 = app back (rev front).
Lemma rev_app : ∀ A (l1 l2 : list A),
    rev (app l1 l2) = app (rev l2) (rev l1).
Lemma rev_list_rect A  : ∀ P:list A→  Type,
    P nil →
    (∀ (a:A) (l:list A), P (rev l) → P (rev (a :: l))) →
    ∀ l:list A, P (rev l).
Theorem rev_rect A : ∀ P:list A → Type,
    P nil →
    (∀ (x:A) (l:list A), P l → P (app l (x :: nil))) →
  ∀ l:list A, P l.
Lemma bisim_list_dlist : Bisimilar ListQueue DListQueue.
Lemma program_independent : ∀ n,
  program ListQueue n = program DListQueue n.
```

The verified Gallina code in Listing 9 was translated to Scala using Scallina. The resulting Scala code is exhibited in Listing 11.

Listing 11. The generated Scala ListQueue program

```
import scala.of.coq.lang._
import Nat._
import Pairs._
import MoreLists._
object ListQueueParam {
  trait Queue {
    type t
    def empty: t
    def push: Nat => t => t
    def pop: t => Option[(Nat, t)]
  }
  object ListQueue extends Queue {
    type t = List[Nat]
    def empty: t = Nil
    def push: Nat => t => t = x => l => x :: l
    def pop: t => Option[(Nat, t)] = l => rev(l) match {
      case Nil => None
      case hd :: tl => Some((hd, rev(tl)))
```

```
    }
  }
  object DListQueue extends Queue {
    type t = (List[Nat], List[Nat])
    def empty: t = (Nil, Nil)
    def push: Nat => t => t = x => { l =>
      val (back, front) = l
      (x :: back, front)
    }
    def pop: t => Option[(Nat, t)] = { l =>
      val (back, front) = l
      front match {
        case Nil => rev(back) match {
          case Nil => None
          case hd :: tl => Some((hd, (Nil, tl)))
        }
        case hd :: tl => Some((hd, (back, tl)))
      }
    }
  }
  def loop[P](op: Nat => P => P)(n: Nat)(x: P): P =
    n match {
      case Zero => x
      case S(n0) => op(n0)(loop(op)(n0)(x))
    }
  def sumElems(Q: Queue)(q: Option[Q.t]): Option[Q.t] =
    q match {
      case Some(q1) => Q.pop(q1) match {
        case Some((x, q2)) => Q.pop(q2) match {
          case Some((y, q3)) => Some(Q.push(x + y)(q3))
          case None => None
        }
        case None => None
      }
      case None => None
    }
  def program(Q: Queue)(n: Nat): Option[Nat] = {
    val q = loop(Q.push)(S(n))(Q.empty)
    val q0 = loop(_ => (q0: Option[Q.t]) => sumElems(Q)(q0))(n)(Some(q))
    q0 match {
      case Some(q1) => Q.pop(q1) match {
        case Some((x, q2)) => Some(x)
        case None => None
      }
      case None => None
    }
  }
}
```

References

1. Amin, N., Grütter, S., Odersky, M., Rompf, T., Stucki, S.: The Essence of dependent object types. In: Lindley, S., McBride, C., Trinder, P., Sannella, D. (eds.) A List of Successes That Can Change the World. LNCS, vol. 9600, pp. 249–272. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_14
2. Appel, A.W.: Verified Functional Algorithms, Software Foundations, vol. 3 (2017). Edited by Pierce, B.C.
3. Guallart, N.: An overview of type theories. Axiomathes **25**(1), 61–77 (2015). https://doi.org/10.1007/s10516-014-9260-9
4. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12251-4_9
5. Hindley, R.: The principle type-scheme of an object in combinatory logic. Trans. Am. Math. Soc. **146**, 29–60 (1969)
6. Hupel, L., Kuncak, V.: Translating scala programs to isabelle/HOL. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 568–577. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_38
7. Klein, G., et al.: seL4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, 11–14 October 2009, pp. 207–220. ACM (2009). https://doi.org/10.1145/1629575.1629596
8. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, 11–13 January 2006, pp. 42–54. ACM (2006). https://doi.org/10.1145/1111037.1111042
9. Letouzey, P.: A new extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39185-1_12
10. Letouzey, P.: Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming: Program extraction within Coq proof assistant), Ph.D. thesis, University of Paris-Sud, Orsay, France (2004). https://tel.archives-ouvertes.fr/tel-00150912
11. Letouzey, P.: Extraction in Coq: an overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 359–369. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69407-6_39
12. The Coq development team: The Coq proof assistant reference manual, version 8.0. LogiCal Project (2004). http://coq.inria.fr
13. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978). https://doi.org/10.1016/0022-0000(78)90014-4
14. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
15. Odersky, M., Rompf, T.: Unifying functional and object-oriented programming with scala. Commun. ACM **57**(4), 76–86 (2014). https://doi.org/10.1145/2591013

16. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-step Guide, 2nd edn. Artima Incorporation, Walnut Creek (2011)
17. Pierce, B.C.: The science of deep specification (keynote). In: Visser, E. (ed.) Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2016, Amsterdam, Netherlands, 30 October–4 November 2016, p. 1. ACM (2016). https://doi.org/10.1145/2984043.2998388