CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Experimental Evaluation of Formal Software Development Using Dependently Typed Languages

Ferenc Tamasi

CISTER-TR-190613

Experimental Evaluation of Formal Software Development Using Dependently Typed Languages

Ferenc Tamasi

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

https://www.cister-labs.pt

Abstract

We will evaluate three dependently typed languages,and their supporting tools and libraries, by implementing thesame tasks in each language. One task will demonstrate the basicdependent type support of each language, the other task willshow how to do basic imperative programming combined withtheorem proving, to ensure both resource safety and functionalcorrectness.

Experimental Evaluation of Formal Software Development Using Dependently Typed Languages

Ferenc Tarnai
Faculty of Engineering
University of Porto
Porto, Portugal
up201809113@fe.up.pt

***Abstract*—We will evaluate three dependently typed languages, and their supporting tools and libraries, by implementing the same tasks in each language. One task will demonstrate the basic dependent type support of each language, the other task will show how to do basic imperative programming combined with theorem proving, to ensure both resource safety and functional correctness.**

***Index Terms*—formal software development, dependent types, Coq, Iris, Agda, Fstar, ST monad, Hoare monad, Dijkstra monad**

I. Introduction

Dependently typed programming is getting some attention in the past years. Noticeable for instance in [2], where prominent researchers in the area state that *"Dependently typed programming languages like Agda are gaining in popularity, and dependently typed programming is also becoming more popular in the Coq community, for instance through the use of some recent extensions."*

The interest is motivated by the need to find the right balance between usability and flexibility when applying the increased accuracy of dependent types in describing program behavior. One can use dependent types in situations ranging from disciplined dynamic typing (Dependent JavaScript [18]), to prove memory correctness of the standard library of a statically typed language (RustBelt [27]), or correctness of a compiler (CompCert [31]).

This growing popularity is also demonstrated by the active tool development to explore working with dependent types, Wikipedia lists 11 actively developed languages with dependent type support.

Three of the tools used in academia are compared, in the context of formal software development. We describe the performed experiment of executing the same tasks in the selected environments.

This paper is structured as follows: in Section II we provide an introduction of dependent types. In Section III, we describe the tool selection process, and a short introduction of the selected tools. In Section IV, we describe the experiments that will be conducted with each of the selected tools. In Section V we describe the implementations of each of the selected tasks with each of the selected tools and explain the experimental results that we have obtained. Finally, in Section VI the conclusions drawn from our experiments are presented, as well as possible future areas of interest.

II. Background

Dependently typed languages [39] extend traditional typed languages, by allowing the types of values to depend on other values. For illustration purposes, we will introduce the concept of dependent types in pseudo C++. *

In standard C++, it is possible to define the type `DepType` as seen in Listing 1, but the `value` template parameter must be available at compile time.

```
class DepType<int value> {};
```

Listing 1: Compile time dependent type in C++

If C++ would be a dependently typed language, we could define the function `pi()` as shown in Listing 2 where the argument is only available at runtime, but the return type depends on the argument's value. Another, more explicit example is the function `pi2()` depicted in Listing 2, where the return types are not versions of the same base type. In `div2` a simple constraint is presented, ensuring that the function can only be called if the returned values are exactly one half of the argument.

```
auto pi(int x) -> DepType<x> {
  return DepType<x>();
}
// return type depends on runtime value of `x`
auto pi2(int x) -> (x ? int : (char const *))
    {
  return x ? 1 : "Hello World!";
}
// only allow calls, if `x` is even
int div2(int x, (x % 2 == 0 ? std::monostate :
     void) x_is_even) {
  return x / 2;
}
// property helper: mapping booleans to types.
// true is a trivially produceable value
// false is a non-produceable value
#define Prop(e) (((e) ? std::monostate : void)
    )
```

Listing 2: C++11 Π

*A similar attempt for refinement types can be found in [48] and [44].

And we could define a `struct` like various `Sigma...` in Listing 3. Here a dependently typed C++ could check, that values of type `Sigma_dependent_type` can only be created if the constraint described in the type of `i_or_s` is satisfied.

A Sigma (Σ) type (also called dependent pair) in a dependently typed language is a structure with two elements, where the *type* of the second element depends on the `value` of the first element.

```
struct Sigma_class {
  int x;
  DepType<x> d;
};

struct Sigma_union {
  int x;
  union {
    short i; // x != 0
    char const * s; // x == 0
  };
};

struct Sigma_dependent_type {
  int x;
  (x != 0 ?
    short :
    char const *
  ) i_or_s;
};
```

Listing 3: C++ Σ

This idea allows us to describe properties of values. In Listing 4, a type is defined, which can only hold even integers. The assurance of evenness of `x` depends on the impossibility of creating a value of type `void`, which is the calculated type of the field `evenness_proof` in the odd `x` case. In case of an even `x`, the calculated type for the proof field is `std::monostate`, a type that has a single possible value, thus has no information content.

We could use any other type instead of `monostate`, but this expresses the intent that we don't care what the value is, as long as it exists (as opposed to the `void` case, where we want to ensure non-existence).

```
struct even_ints {
  int x;
  (x % 2 == 0 ? std::monostate : void)
      evenness_proof;
};
```

Listing 4: Even integers

This sort of constrained types are called refinement types [22] their general form in our pseudo dependent C++ is shown in Listing 5. (Which itself is a specialized form of the Σ types from Listing 3).

```
template <typename T, bool (*P)(T)>
struct refined_T {
        T v;
        (P v ? std::monostate : void) proof;
};
```

Listing 5: Dependent C++ refinement types

The analogy of "a `value` to its *type*, is what a `proof` is to it's *logical formula*", is described as the Curry-Howard correspondence [26]. In our pseudo C++ the type of the proofs are always either `void` or `std::monostate`, depending on the condition's value in the ternary expression. In proper dependently typed languages on the other hand, logical formulas are themselves types. For example, a conjunction of two formulas is the type, that has two type parameters, and in order to create a value (that is a proof of the conjunction), one has to provide two values, each with a corresponding type.

```
template <typename A, typename B>
struct Conjunction {
        Conjunction(A, B) {}
};
template <typename A, typename B>
struct Disjunction {
        static Disjunction left(A a) {}
        static Disjunction right(B b) {}
};

/* using the property helper macro from above,
   a refinement type expressing, that an
       integer
   is within the specified range. */
template<int low, int high>
struct Range {
        int x;
        Conjunction<Prop(low <= x), Prop(x <
            high)> proof;
};
```

Listing 6: Dependent C++ logical connectives

For a more complete illustration Listing 7 a sorted linked list implementation is shown in dependent C++.

```
struct List {
  int v;
  List * next;
};
/* struct sortedlist: a type expressing that a
     List, starting
  from node, is sorted */
struct SortedList {
  List * node;
  SortedList * nextproof;
  /* property type expressing sortedness,
    depends on values: node and nextproof. */
  Disjunction<
    Prop(node == nullptr),
    Disjunction<
      Prop(node->next == nullptr),
      Conjunction<
        Prop(node->v <= node->next->v),
        Conjunction<
          Prop(nextproof != nullptr),
          Prop(nextproof->node = node->next)
        >
      >
    >
```

```
  > proof_value; // the proof_value field is
      compile time only
};
```

Listing 7: C++ sorted list

Building up proof terms is similar to calculating traditional values, see in Listing 8 as we build up `proof_value`.

```
SortedList *
prepend(
  int v,
  SortedList * l,
  Prop(l == nullptr || v <= l->node->v)
      v_proof
)
{
  List * node = new List{v: v, next: l ? l->
      node : nullptr};
  SortedList * res = new SortedList{
    node: node,
    nextproof: l,
    proof_value: node == nullptr ? Disjunction
        <...>::left(...) : Disjunction<...>::
        right(...)
    /* the developer builds up a value of the
        type
       specified above, and the compiler
       checks the validity of the types */
  };
}
```

Listing 8: C++ sorted list prepend

III. Tool Selection

The search term `TITLE-ABS-KEY("dependent* type*" AND imperative)` returned 40 hits at Scopus, of those papers 25 are unique. From the unique papers, we selected those that were not introducing a language, but using the language as a tool, in order to select languages that the community found useful in research.

This selection criteria resulted in Agda [51] used in [1], Coq [6] used in [37], [24], and [46], and F* [8] used in [11] [12].

A. Selected Tool Short Introduction

a) Agda

Agda is the name of both the dependently typed functional programming language, and the interactive proof assistant to work with the language, based on typed holes [41] implemented as an Emacs mode.

Agda is based on intuitionistic type theory [38], a foundational system for constructive mathematics. We examined version 2.5.4.1, with stdlib version 0.17.

b) Coq

Coq is the name of a proof management system. It is built on three languages, *Gallina* the dependently typed functional language, *Vernacular* the proof engine management language, and L_{tac} the language for proof tactics. There are multiple interactive environments developed for Coq, the official is CoqIDE, but ProofGeneral for Emacs is also popular.

Coq is also based on intuitionistic type theory. We examined version 8.8.2.

*c) F**

F* (pronounced F star, sometimes written as F$\star$) is a general-purpose functional programming language, with support for program verification, based on dependent types. F* though supports dependent types, it is mainly focused on the refinement type subset.

F* does not make a statement about its foundational logic. We examined version 0.9.6.0.

B. Quick Comparison

The selected tools are all based on languages that support dependent types. The syntax of each language is described in the following resources: Agda [40], Coq [7], and F* [9].

To get an overview of how each language looks like, in the three listings below the same function is defined three times. The function takes a natural number as a parameter and returns a dependent pair as a result.

The result pair's first element in the function body is always set to zero (this is to simplify the example), and the type of the result pair's second element depends on both the function parameter's value (`x`), and the pair's first element's value (`y`). (Since we always set the first element to zero, this is effectively a comparison of the function parameter with zero). The second element's type is either the unit type, or boolean, depending on the comparison result.

Agda:

```
open import Data.Bool using (Bool;
    if_then_else_)
open import Data.Nat using (ℕ; suc; zero; _≟_)
open import Data.Product using (∃; _,_)
open import Data.Unit using (⊤)
open import Relation.Nullary.Decidable using (
    ⌊_⌋)

pi : (x : ℕ) → ∃ (λ (y : ℕ) → (if ⌊ x ≟ y ⌋
    then ⊤ else Bool))
pi zero = (zero , ⊤.tt)
pi (suc _) = (zero , Bool.true)
```

Coq:

```
Require PeanoNat.

Definition pi (x : nat) : { y : nat & if
    PeanoNat.Nat.eq_dec x y then unit else
    bool } :=
  match x return { y : nat & if PeanoNat.Nat.
      eq_dec x y then unit else bool } with
  | O => existT _ O tt
  | S x' => existT _ O true
  end.
```

F*:

```
module PiSigma

val pi : x:nat -> Tot (y:nat & (if x = y then
    unit else bool))
let pi x =
  match x with
```

```
  | 0 -> (|0, ()|)
  | _ -> (|0, true|)
```

Listing 9: ΠΣ in three languages

From this short syntax comparison it is already visible, that the tools take different approaches: in Agda we need to import even the most basic definitions, while in F* we don't need to import anything; Agda typically uses Unicode symbols, while the others use ASCII names. This is only a convention of the developers of the tools, as both F* and Coq has the ability to work with Unicode characters. A library for Coq called Iris [28] for example employs Unicode extensively.

IV. Tasks

We selected two tasks to implement, that represent two areas of functionality:

A. Theorem Proving

Prove the commutativity of addition over the language's default natural ($\mathbb{N}$) type*. That is, for all $a, b \in \mathbb{N}$, the equality $a + b = b + a$ holds. This task exercises the basic theorem proving machinery in the language.

B. Imperative Programming using In Memory Datastructures

Sort an in memory array of fixed size integers. This task demonstrates the language's prowess in combining safe memory management and proving application level properties [52].

Ensuring valid memory addressing is one important use case of dependent types. This problem is mostly mitigated by the hardware getting fast enough to afford runtime bounds checks, and the compilers getting clever enough to elide most of the runtime bound checks†. So this task aims to demonstrate the other important feature of dependent types: the ability to describe high level requirements and certify their implementation (in this case sortedness).

V. Implementation and Results

A. Theorem Proving

1) Agda

a) getting started

Agda is popular enough, that an internet search led us to a partial solution of this problem ‡.

As Agda does not autoload even the most basic definitions, it takes some time to discover, where a definition is located in the standard library. Also, if one wishes to write idiomatic Agda, and the location of a definition is not the canonical way to import a symbol, one has to chase down the wrapping module, that imports, then re-exports the original definition.

b) ergonomics

Agda has an Emacs mode §, where one can use a hole based development style. To create a hole, one enters a question mark (`?`) in place of an expression. The editor then creates a hole context, in which the developer can interactively build up the expression with type the hole requests.

The hole context provides an overview of what values of what types are available, and what is the type of the expression the developer needs to create.

2) Coq

a) getting started

Since the author is quite familiar with Coq already, we had to try to rely on intuition and memory to try to evaluate the starting out experience of Coq.

Coq has a very steep learning curve, but since it is a very mature project, there are plenty of tutorials online, and the tooling is rather featureful and stable.

A similar problem to Agda of standard library discoverability exists in Coq as well, but the situation is improved by the integrated `Search` commands ¶, which find in the current context facts about types or functions.

b) ergonomics

Coq is the tool of the three reviewed, that has the most mature proof facilities.

Coq is designed around interactive proof development, which is similar to the hole based approach of Agda, but it not only provides the context for the developer, but also gives tools to transform the goal and the available values in the context.

When using the interactive facilities, the author proceeds, and issues tactics ‖, that transform the hole, introduce new facts to the context, split the target into parts, for a full list see the Coq Reference Manual.

The implementation presented in `PlusComm.v` is written in the interactive proving style.

c) non-interactive proving

To provide a more direct comparison, we proved the commutativity in the direct style of Agda and F* in `PlusCommDirect.v`. This leads to a very similar proof as with the other tools. One gives a fully formed proof to the language for checking, with no help from the tool.

3) F*

a) getting started

Simple proofs like this can be discharged with the integrated Z3 [19] satisfiability modulo theories solver. F* by convention uses refinement types, in particular the refinement of the unit type, to represent properties. F*, though does not encourage it, is also able to express the original properties-are-types idea of dependent types.

b) ergonomics

F* also has an Emacs mode, that is the recommended way of editing F* sources, called fstar-mode **. It is still in

The default varies between languages, in F it is a refined type, limiting a base type to non-negative values, in Agda and Coq it is a Peano numeral.

†See for example Java, Python, or Rust.

‡https://stackoverflow.com/questions/52282786/proving-commutativity-of-addition-in-agda

§`elpa-agda2-mode` package in Debian

¶https://coq.inria.fr/refman/proof-engine/vernacular-commands.html#coq:cmd.search

‖https://coq.inria.fr/distrib/current/refman/coq-tacindex.html

**https://github.com/FStarLang/fstar-mode.el

early development phase, so some features are not working. Most problematic is the environment's reluctance to work with incomplete source, which is quite the common occurrence during programming.

One useful technique to deal with this limitation is using the `admit()` function in place of the missing expressions in the code. This is similar to Agda's hole oriented programming, but it does not provide the helpful interactive context, but helps with the partial source problem.

c) F task with Peano numbers*

Since the task following the original description was so quickly and smoothly solved by F*, we decided to include the task implemented for Peano [42] numbers, using both the refined-unit-as-prop approach in `PlusCommPeano.fst`, and an explicit type-as-prop in `PlusCommPeanoProp.fst`. During the implementation of these solutions, the immaturity of the proof development environment forcing us to provide the solutions without support of the tool was a little bothering, but peeking at the Agda solution helped the proof along.

The solution in `PlusCommPeano.fst` still relies on the built-in Z3 automation. Since we are not using the built in numerical types the proof itself shows a little more of the internals.

The solution in `PlusCommPeanoProp.fst` is managing the proof terms explicitly, and it seems this method of proving disables the built-in proof automation, as the full proof term had to be entered.

Even though F* supports proof automation through tactics, since these are not interactive, they don't help when such a small scale task is developed. But we expect, in more complicated tasks (e.g. in a domain specific language implemented in F*), they can be quite useful.

B. Imperative Programming using In Memory Datastructures

1) ST&Hoare introduction

One way of handling stateful computation is through the ST monad [35] introduced in Haskell. The ST monad provides primitives to work with the heap, but it prevents direct access to the memory. In fact the ST monad, effectively hides the values that are in memory from the host language. The established way to workaround this, is to use Hoare logic [25].

In the Hoare monad the ST monad is enriched with pre- and post-conditions around ST operations. This enriched construct is called the Hoare triple. It consists of: the *pre-condition*, which specifies the requirements about the environment for when the *action* is enabled; the ST *action* which defines the operation to be performed; and the *post-condition*, which specifies the guarantees after the ST operation is performed, based on the values in the heap both before and after the operation, as well as the value generated by the ST operation.

A newer structure, called the Dijkstra monad [47], is also used, which fulfills the same function as the Hoare monad, but instead of pre- and post-conditions, it uses weakest precondition predicate transformation [20]. A weakest precondition (WP) predicate transformer generates a pre-condition, based on a post-condition, that is the least restrictive pre-condition, that enables the execution of the ST action.

2) Agda

a) ST in Agda

Unfortunately Agda does not include the ST monad in the standard distribution, so we used an implementation from [32]. In [32] Kovács models what in Haskell is called STRef [10], but limiting the supported types to boolean and natural numbers. It doesn't support monadic bind operation either, so we had to resort to continuation passing [5]

b) modal logic

Since in order to reason about the changes in the ST heap, we would need some sort of modal logic [36] over the values stored in the heap (to be able to talk about before/after values). But Hoare logic is not included in the implementation of Kovács's ST implementation, so we abandoned the attempt of proving the sortedness of the resulting list.

We settled for only showing how to work with memory in the imperative style, and only giving guarantees about the validity of indexing in the array (we could do this, since the indexing happens in the host language), not about the sortedness of the result (which would require access to the values stored in heap memory).

3) Coq

a) ST in Coq

Coq does not include imperative features in its standard library. Since Ynot [17] was used in [24] as the library implementing mutable state, we first tried to use that, but we found, that it has been abandoned since 2014, and does not compile with the latest Coq. An actively developed similar library for Coq is Iris [28]. We examined Iris development version with Git hash 455fec93.

Iris has a larger scope, namely it also targets concurrent programs, but in contrast to Ynot, Iris does not support compiling the program to executable format (called extraction in Coq [*]). This follows from the fact, that Ynot uses shallow embedding and Iris uses deep embedding.

Both Ynot and Iris weaken the Coq guarantees, by introducing the possibility of creating non terminating programs, which are disallowed by vanilla Coq.

b) modal logic

Iris is based on concurrent separation logic [33] we will use the instantiation of the base logic for memory heaps. The implementation uses the Dijkstra [47] monad is based on weakest precondition transformation [20], as opposed to the Hoare monad, that is based on pre- and post-conditions [25]. In practice, since the pre- and post-conditions are more natural to think about, the predicate transformers of weakest precondition calculus is hidden from the developer, and the predicate transformer is generated from the provided pre- and post-conditions.

c) ghost variables

Iris uses ghost variables to help express properties of the program. Ghost variables *can not* interact with the evaluation

[*]https://coq.inria.fr/distrib/current/refman/addendum/extraction.html

of the program, they are only present while proving program properties.

The ghost variable is connected to the real variables through properties. It is said, that a ghost variable models a real variable. For example in this task, we are modeling an array, using a pointer as real variable, and a list as ghost variable, expressing, that the pointer points to a value that is equal to the value of the first element of the list, the (pointer+1) points to a value that is equal to the second element of the list, $\forall i : \mathbb{N}, i < |list| \implies pointer + i \mapsto list!!i$

d) proof management

Coq itself is an interactive proof assistant, so the basic mode of operation is building proofs, by interactively applying tactics that transform the goal [*].

Coq also supports proof automation, which involves automated proof term generation, and proof search for fitting terms. Iris uses this facility and the typeclass system of Coq extensively, creating a fourth and fifth language on, top of the three languages already in Coq. **Iris logic**, a DSL implementing an affine Concurrent Separation Logic (CSL) [13] . And **Iris Proof Mode**, a tactic language to deal with proofs in Iris logic [34].

e) discoverability

Coq itself is well established and well documented, with many tutorials to choose from [†].

Iris on the other hand is still under active development (2.0 released in 2016, 3.0 in 2017), finding the relevant documentation is challenging, and sometimes the relevant documentation does not exist (c.f: [29] chapter 1.3).

f) location arithmetic

The base logic does not define arithmetic operations for locations (pointers), so for demonstration purposes we added an indexing extension `location_arithmetic`.

A proper location arithmetic should take into account the size of the allocation, but for simplicity, we defined an array as elements separated by one "unit" of whatever an increment of a location value by one means, as this is not material to the meaning of the proof , but simplifies the proofs themselves.

g) numeric conversions

Locations are represented as `positive` ($\mathbb{Z}^+$) numbers, the standard library mostly uses $\mathbb{N}$, and the default number type is $\mathbb{Z}$.

The interaction of these three number types creates a huge time sink, as the usual rules of mathematics do not apply anymore. The built in conversion from `nat` ($\mathbb{N}$) type maps $0_{\mathbb{N}}$ to $1_{\mathbb{Z}^+}$. This means, for example, that depending on whether we convert the arguments, or the result of an addition, we get different results: $(0_{\mathbb{N}} +_{\mathbb{N}} 0_{\mathbb{N}})_{\mathbb{Z}^+} \neq (0_{\mathbb{N}})_{\mathbb{Z}^+} +_{\mathbb{Z}^+} (0_{\mathbb{N}})_{\mathbb{Z}^+}$, the left side is 1 the right side is 2.

h) fun with separation logic

Separation logic is a mixture of linear and nonlinear logic [3], which for us means, that facts about a variable in the liner logic part can only be used once.

Proving with separation logic used by Iris compared to the standard intuitionistic logic used by Coq, forces a more disciplined approach to proving, which at first does pose some difficulties, but it also helps offload some mental burden from the developer to the compiler [14].

In this task keeping track of memory resources is solved by the affine logic perfectly (as it was designed to do). [‡]

i) predetermined heap types

The CSL DSL only supports a pre-determined list of types [§]. This limited the sorting predicates as well, since only the operators in the DSL can be used.

j) proof length

As we get to higher level operations, the associated proofs get shorter, this gives a probable explanation, why large projects use Coq (RustBelt [27], CompCert [31], Fiat-Crypto [21], VST [4]).

*4) F**

*a) ST in F**

F* uses algebraic effects [43] for modeling stateful computations. F* implements the ST effect in its standard library.

b) discoverability

We found the discoverability of the F* libraries lacking, but once we settled to base the implementation on `examples/algorithms/QuickSort.Array.fst` from the F* source distribution, the standard library turned out very well equipped to deal with sorting.

c) modal logic

F* also uses the Dijkstra monad [47] to keep track of the programs environment like Coq+Iris.

d) proof management

F* relies on implicit proofs, generated from the provided preconditions proving the post-conditions. This makes the proof process opaque, and in case the goals are not discharged, an exercise in guessing what the automated proof machinery wants as input, to be able to find the solution, and which knobs of the magic machine has to be tweaked to help it through the proof search.

The approach we took was, to throw more and more facts at the proof search, and once it succeeds, start removing the ones, that keep the goals discharged. Not a very efficient, scalable, or dignified way to work. But alas no alternative exists, barring one becomes intimately familiar with the internal proof searching algorithms of F*. Then repeat the exercise for the next F* releases, ad infinitum.

e) array lib

F*'s array implementation [¶] can not track individual cell modifications with the `modifies` utility of ST, only the whole array can be declared as modified with the `modifies` keyword.

[*]The whack-a-mole style proving http://gallium.inria.fr/blog/coq-eval/

[†]https://coq.inria.fr/documentation

[‡]Brady in [16] demonstrates the usefulness of linear logic (a slightly stricter version of affine logic) in the context of **Id**ris. Brady presenting this can be found here: https://www.youtube.com/watch?v=mOtKD7ml0NU&t=30m53s)

[§]boolean, $\mathbb{Z}$, unit, location (pointer), prophecy (which seems to be an internal type)

[¶]`FStar.Array`

C. Quantitative Analysis

a) source metrics

In Table I we are showing a numerical evaluation of the size of our solutions. Columns task 1 and task 2 refer to the number of lines in the solution for task 1 and 2 respectively, counting all non-comment lines.

In columns body 1 and body 2, we show the number of lines, with comments and import statements removed, while in core 2 we show the number of lines directly related to sorting and the sortedness proof, excluding the generic proofs that should be added to either the standard library of the tool, or the array library.

TABLE I: Number of lines per task per tool

	task 1	body 1	task 2	body 2	core 2
Agda	48	45	123*	80*	26*
Coq	12	12	1433	1265	574
F*	3	3	109	90	65

*: The agda solution for the second task only proves memory safety, not sortedness.

Agda has the biggest overhead associated with library imports (the difference between the task and body columns are 18% and 30%). This is a blessing and a curse at the same time, as it makes writing the code more tedious, but the one reading the code is helped by the explicit dependency enumeration.

In absolute numbers the Coq solution is an order of magnitude larger than the other two solutions. This is only a fair comparison against F*, but it still shows that proof search in F* does work*.

b) development time

The rough approximation of the time to solve tasks 1&2: Agda 6 days, Coq 7 days, F* 3 days. These numbers are based on the version control history of the author. Here too, Coq is the one requiring the most time, even though the author has the most experience with it.

VI. Conclusion

The three languages take very different approaches to present the power of dependent types to the user. Thus it is impossible to declare a best tool, but we will describe the situation in which each tool excels.

a) history

Coq is the oldest of the three, with many successful industrial [27] [31] and academic [23] projects under its belt.

Agda is also established, especially in programming language research [2] [30].

F*, a relatively recent development coming from Microsoft, discourages creating proof terms by hand, presumably to appeal to users who wish to avoid dealing with the minutia of proofs. This is great as long as the user can stay within the confines of the F* design, but at the cost of a sudden increase in discomfort, if one must leave the beaten path.

Based on their history, Coq can be considered the standard tool, when one wishes to work with something that "everybody else" uses, and a tool that will probably be around later.

b) proving

Coq uses interactive tactics to prove goals, which is very convenient, but may lead to large proof scripts in case one does ad-hoc proofs. But Coq also has the tools to make the proofs concise, provided one works in a fixed domain, and creates the necessary abstractions. Iris for example embedded the full logic of CSL and tactics to work with it in Coq.

Agda is more in line with traditional programming languages, as it expects the user to write the expressions that will produce the expected value.

F* does not want the user to prove anything, it only expects enough facts to be presented, so that the built-in prover can work out a proof.

The choice of tool depends very much on the task one wishes to solve. F* works great, as long as one can fit the task at hand into what F* can work with, and one is willing to do the guesswork involved in trying to work out what the missing piece might be for the automated prover to go through.

Coq and Agda both provide interactive proving environments, but the larger user base and longer history of Coq give an edge that Agda can't compete with.

c) messy requirements vs messy proofs

As we stated in the previous paragraph, F* discourages user proofs, but this makes requirements unnecessarily large, since the automated tool needs a lot more detail, than what a human prover needs to prove the same goal.

If one can fit one's work in F*'s beaten path, then it works great, otherwise Coq or Agda is probably a better choice as they provide a more natural environment to create proof terms.

d) tactics generated vs hand crafted proofs

In [50] Wadler states *"Proofs in Coq require an interactive environment to be understood, while proofs in Agda can be read on the page."*, while this is true for the languages themselves, but Proviola [49] can alleviate this problem of Coq, by recording the proof state after each tactic execution, and producing an html document with the proof state added for each tactic. F* does not have this problem, as the proof terms do not appear either in the source, or during proving.

Whether it is easier to read complete proof terms, or the replay of a step by step creation of a proof term is dependent of the task at hand, but the author thinks, that it is more straightforward to create scripts step by step in Coq, though it does require discipline on the programmer's part, so as to not create a write-only script †.

1) future work

Both the breadth and the depth of this work could be extended. Doing the same tasks in other, less established languages like Idris [15], or ATS [53], or trying different libraries like FCSL [45].

The depth increased by adding more interesting tasks, for example, investigating the generation of verified executables

At least for this case, after appeasing the search machinery. It would be interesting to see in a larger project, with not so straightforward properties, how would the built in logic of F behave?

†http://www.jargon.net/jargonfile/w/write-onlylanguage.html

from the verified sources, or comparing how different tools enable verifying resource management other than memory (files, network sockets, etc), or verifying non functional requirements like security or real time constraints.

References

[1] Stephan Adelsberger, Anton Setzer, and Eric Walkingshaw. Developing gui applications in a verified setting. Cham, 2018.

[2] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. $\pi\sigma$: Dependent types without the sugar. In *International Symposium on Functional and Logic Programming*, 2010.

[3] Andrew W Appel. Tactics for separation logic. *INRIA Rocquencourt and Princeton University, Early Draft*, 2006.

[4] Andrew W Appel. *Program logics for certified compilers*. Cambridge University Press, 2014.

[5] Andrew W Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–302. ACM, 1989.

[6] Coq Authors. https://coq.inria.fr/, 2018.

[7] Coq Authors. Documentation — the coq proof assistant. https://coq.inria.fr/documentation, 2018.

[8] F* Authors. https://fstar-lang.org/, 2018.

[9] F* Authors. F* tutorial. https://www.fstar-lang.org/tutorial/, 2018.

[10] Haskell Wiki Authors. Data.STRef. https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-STRef.html.

[11] Karthikeyan Bhargavan et al. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *n/a*, Dagstuhl, Germany, 2017.

[12] Karthikeyan Bhargavan, Cedric Fournet, and Markulf Kohlweiss. mitls: Verifying protocol implementations against real-world attacks. *IEEE Security & Privacy*, 14(6):18–25, 2016.

[13] Lars Birkedal and Aleš Bizjak. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic*. 2018.

[14] Rúnar Bjarnason. Maximally powerful, minimally useful. http://blog.higher-order.com/blog/2014/12/21/maximally-powerful/, 2014.

[15] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. https://www.idris-lang.org/, 2013.

[16] Edwin Brady. *Type-driven development with Idris*. Manning Publications Company, 2017.

[17] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. *ACM Sigplan Notices*, 44(9):79–90, 2009.

[18] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for javascript. *ACM SIGPLAN Notices*, 47(10):587–606, 2012.

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[20] Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[21] Andres Erbsen. *Crafting certified elliptic curve cryptography implementations in Coq*. PhD thesis, Massachusetts Institute of Technology, 2017.

[22] Tim Freeman. Refinement types ml. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1994.

[23] Georges Gonthier. A computer-checked proof of the four colour theorem, 2005.

[24] Colin S Gordon, Michael D Ernst, and Dan Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *ACM SIGPLAN Notices*, volume 48, pages 73–84. ACM, 2013.

[25] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[26] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[27] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the rust programming language. 2017.

[28] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ACM SIGPLAN Notices*, volume 51, pages 256–269. ACM, 2016.

[29] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[30] Wolfram Kahl. Dependently-typed formalisation of relation-algebraic abstractions. In *International Conference on Relational and Algebraic Methods in Computer Science*, pages 230–247. Springer, 2011.

[31] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. Closing the gap – the formally verified optimizing compiler CompCert. CreateSpace, 2017.

[32] András Kovács. Computing ST monad in vanilla Agda. https://gist.githubusercontent.com/AndrasKovacs/07310be00e2a1bb9e94d7c8dbd1dced6.

[33] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*, pages 696–723. Springer, 2017.

[34] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. *ACM SIGPLAN Notices*, 52(1):205–217, 2017.

[35] John Launchbury and Simon L Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN Notices*, volume 29, pages 24–35. ACM, 1994.

[36] Clarence Irving Lewis and Cooper Harold Langford. *Symbolic logic*. 1932.

[37] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with i/o. *Journal of Symbolic Computation*, 46(2):95, 2011.

[38] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[39] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's type theory*, volume 200. Oxford University Press, Oxford, 1990. Out of print.

[40] Ulf Norell. Dependently typed programming in agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

[41] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A Hammer. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages*, 3(POPL):14, 2019.

[42] Giuseppe Peano. *Arithmetices principia: nova methodo exposita*. Fratres Bocca, 1889.

[43] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

[44] reddit user denito2. So i translated part of the 1st chapter of adam chlipala's book code from coq into c++ template metaprogramming... https://godbolt.org/z/bZTZrK, 2019.

[45] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. https://software.imdea.org/fcsl/, 2015.

[46] Gordon Stewart, Anindya Banerjee, and Aleksandar Nanevski. Dependent types for enforcement of information flow and erasure policies in heterogeneous data structures. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, pages 145–156. ACM, 2013.

[47] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Notices*, volume 48, pages 387–398. ACM, 2013.

[48] Marco Syfrig. Dependent types: Level up your types. https://eprints.hsr.ch/577/1/MarcoSyfrigDependentTypes_eprints.pdf, 2016.

[49] Carst Tankink, Herman Geuvers, James McKinna, and Freek Wiedijk. Proviola: A tool for proof re-animation. In *International Conference on Intelligent Computer Mathematics*, pages 440–454. Springer, 2010.

[50] Philip Wadler. Programming language foundations in agda. In *Brazilian Symposium on Formal Methods*, pages 56–73. Springer, 2018.

[51] The Agda wiki. http://www.cs.chalmers.se/~ulfn/Agda, 2018.

[52] Hongwei Xi. Programming with dependently typed data structures. 1999.

[53] Hongwei Xi. Applied type system: An approach to practical programming with theorem-proving. http://www.ats-lang.org/, 2017.