Extracting the Power of Dependent Types

Artjoms Šinkarovs
Heriot-Watt University
Edinburgh, Scotland, UK
a.sinkarovs@hw.ac.uk

Jesper Cockx
TU Delft
Delft, Netherlands
j.g.h.cockx@tudelft.nl

Abstract

Most existing programming languages provide little support to formally state and prove properties about programs. Adding such capabilities is far from trivial, as it requires significant re-engineering of the existing compilers and tools. This paper proposes a novel technique to write correct-by-construction programs in languages without built-in verification capabilities, while maintaining the ability to use existing tools. This is achieved in three steps. Firstly, we give a shallow embedding of the language (or a subset) into a dependently typed language. Secondly, we write a program in that embedding, and we use dependent types to guarantee correctness properties of interest within the embedding. Thirdly, we extract a program written in the original language, so it can be used with existing compilers and tools.

Our main insight is that it is possible to express all three steps in a single language that supports both dependent types and reflection. Essentially, this allows us to express a program, its formal properties, and a compiler for it hand-in-hand, offering a lot of flexibility to programmers. We demonstrate this three-step approach by embedding a subset of the PostScript language in Agda, and illustrating it with several short examples. Thus we use the power of reflection to bring the benefits of dependent types to languages that had to go without them so far.

CCS Concepts: • **Software and its engineering** → *Source code generation*; *Translator writing systems and compiler generators*; *Software verification*; *Domain specific languages.*

Keywords: embedded languages, program extraction, program verification, dependent types, reflection, Agda

ACM Reference Format:
Artjoms Šinkarovs and Jesper Cockx. 2021. Extracting the Power of Dependent Types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3486609.3487201

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

GPCE '21, October 17–18, 2021, Chicago, IL, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9112-2/21/10.
https://doi.org/10.1145/3486609.3487201

1 Introduction

It is often desirable to guarantee that a certain class of errors does not occur in a given program. These error classes may include: dereferencing a null pointer, division by zero, or out of bound indexing. In most programming languages, the properties that can be checked statically are limited. Nevertheless, moving to a different language is not always an option. For example, these languages may come with great tooling, compilers and/or libraries. In this paper, we investigate how to provide strong static guarantees for programs in a given language, while maintaining the ability to to use existing compilers and tools.

One powerful method for enforcing almost any functional property of a program statically is by using *dependent types*, as used for example in Coq [Coq Development Team 2021] or Agda [Agda Development Team 2021]. In a dependently typed language, properties can be encoded as types, and if the program typechecks, the property is guaranteed to hold. If we extend our existing language with dependent types (for the verification part) and then eliminate them (for using the original compiler), then our goal is achieved.

While it is possible to add dependent types to an existing language from scratch, doing so is time-consuming and error-prone. Instead, we can *embed* the language we want to use into an existing dependently typed host language, which allows us to re-use the typechecker of the host language to typecheck the embedded language.

There are two common approaches to language embedding: deep and shallow. Shallow embeddings define languages in terms of functions and operators of the host language. Programs in shallowly embedded languages are programs in the host language, so they are immediately runnable; types of the embedded language are types of the host language, so the host typechecker is used for checking embedded programs. However we cannot easily get the syntactic representation of a shallowly embedded program.

Meanwhile, a deep embedding defines a data type to represent the AST of the embedded language. This requires us to define a type system and evaluator for the language separately. With a dependently typed host language the AST and the type system can be defined as a single data structure, which is typically referred to as a tagless interpreter [Carette et al. 2007; Pasalic et al. 2002]. The benefit of deep embedding is that we get full access to the syntactic structure of the embedded program. This approach works beautifully for embedded languages with simple type systems. However,

for embedded languages with dependent types, tagless interpreters are technically possible [Chapman 2009; Danielsson 2007], but are *extremely* challenging to define and use in practice. In particular, the mutual dependencies between the syntax for contexts, types, and terms and the definitional equality of the object language require large and complex inductive-inductive or inductive-recursive types to encode [Chapman 2009; Danielsson 2007; McBride 2010]. Even if we manage to give a faithful deep embedding of the language, actually writing programs in it is still an arduous task compared to writing programs in the meta-language directly.

The key insight of this paper lies in proposing a novel approach that combines the best parts of shallow and deep embeddings through the use of reflection in a dependently-typed host language such as Coq, Agda, Idris, or Lean.

Our main objective of the paper is to demonstrate that the proposed approach works in practice. Therefore, we guide the reader through the entire process of: i) shallow embedding of the target language; ii) writing programs in this embedding; iii) using dependent types to statically enforce the properties of interest; iv) extracting a program in the syntax of the target language from it. We run the extracted program using existing tools and libraries. Essentially, we demonstrate that a shallow embedding in a host language with reflection suddenly becomes deep.

To keep presentation as easy as possible to follow, we have chosen a simple target language to embed — we use a subset of PostScript [Adobe Systems Incorporated 1999], a stack-based language used in the corresponding document format. This paper is not concerned with practical application of the presented embedding, so we consciously keep the setup as minimal as possible. It is powerful enough to motivate the use of dependent types *within* the embedding, and to demonstrate the essence of the proposed approach. While we include many patterns found in bigger languages, we leave the scaling of the approach out of the scope of the paper referring the interested reader to [Sinkarovs and Cockx 2021] where we work out such generalisations.

Our ***main contributions*** are as follows:

- We propose a novel approach that can be used to statically enforce almost any property of programs in an existing language, by embedding it in a host language that supports dependent types and reflection.
- We demonstrate this approach by a concrete example: an embedding of a subset of the PostScript language into Agda (Sect. 3). We give several examples of PostScript programs written in our embedding, and demonstrate how to statically enforce properties such as the absence of stack underflows, termination, and functional correctness.
- We show how to use reflection in Agda to implement an extractor that translates code written in our embedding to plain PostScript (Sect. 4).
- We show how to our approach provides easy support for partial evaluation by making use of normalisation in the host language. This allows us to use functions of the host language as macros, and optimise porgams via rewrite rules (Sect. 5).

When dealing with embedded languages that are extracted and run by external compilers, we are fundamentally bound to have potential semantic mismatches. Semantics of the embedding may differ from the semantics implemented by external compilers, and the translation from the embedding back to external compilers can also be faulty. Real world compilers face exactly the same problem: how do you know that generated assembly code is doing what you think it is doing? Addressing such problems requires fully verified toolchains that run on a verified hardware. Enormous research efforts are spent towards addressing this problem [Kumar et al. 2014; Leroy 2009], but this is beyond the scope of this paper.

This entire paper is a *literate Agda* script[1]: all code is type-checked while generating the paper.

2 Background

Agda is an implementation of Martin-Löf's dependent type theory [Martin-Löf 1998] extended with many constructions such as records, modules, do-notation, *etc.* We start with a brief overview of key Agda constructions that are used in this paper. We also present relevant parts of the reflection API. For a more in-depth introduction to Agda refer to the Agda user manual [Agda Development Team 2021].

Datatypes. Datatypes are defined as follows:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

```
data Vec (A : Set) : ℕ → Set where
  []   : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n
       → Vec A (suc n)
```

The type ℕ of unary natural numbers is a datatype with two constructors: zero and suc. The usual notation 0, 1, 2, ...is implicitly mapped into ℕ. The type ℕ itself belongs to the type Set, Agda's builtin type of all (small) types.

Agda allows the declaration of indexed datatypes[2], such as the type Vec which is indexed over values of type ℕ. The type Vec A n represents vectors holding n values of type A. It has two constructors: [] for the empty vector of length zero and _::_ for adding an element to a vector, increasing the length by 1. Curly braces indicate hidden arguments that can be left out at function applications.[3] The underscores in the name of _::_ indicate mixfix syntax:[4] we can write x :: xs instead of _::_ x xs.

[1] The sources of the paper can be found at https://github.com/ashinkarov/agda-stacklang.

[2] agda.readthedocs.io/en/v2.6.2/language/data-types.html

[3] agda.readthedocs.io/en/v2.6.2/language/implicit-arguments.html

[4] agda.readthedocs.io/en/v2.6.2/language/mixfix-operators.html

Pattern matching. Functions are defined in a pattern-matching style:

```
_+_ : ℕ → ℕ → ℕ
zero    + y = y
(suc x) + y = suc (x + y)
```

```
tail : {n : ℕ} →
  Vec ℕ (suc n) → Vec ℕ n
tail (x :: xs) = xs
```

Agda requires that all definitions by pattern matching cover all cases. In the definition of tail, we omit the case for the empty vector [] because it takes an input of type Vec *A* (suc *n*), so it can never be called with input [].

Termination checking. To ensure totality, Agda checks that all recursive functions are terminating on all inputs.[5] While it is impossible to infer termination for an arbitrary function due to the halting problem, the termination check of Agda is powerful enough to handle many common cases. The main idea behind the check is that at least one of the arguments to the function has to become structurally smaller than the input argument in each recursive call. For example, in the recursive call to _+_, the first argument is *x*, which is structurally smaller than suc *x*.

Proving equalities. Agda is both a programming language and a proof assistant. One common example of this is the equality type _≡_ that expresses equality of its two arguments. It has a single constructor refl : *x* ≡ *x* stating that any value *x* is equal to itself. Using the equality type, we can state and prove equations between Agda expressions, which are then checked by the typechecker. For example, we can prove that 1+1 = 2 by refl : 1 + 1 ≡ 2. Although in this paper we only prove a few basic properties, the possibility to prove arbitrary (functional) properties of programs embedded in Agda is an important benefit of our approach.

Run-time irrelevance. Function types can be marked as *run-time irrelevant* [McBride 2016] with the @0 annotation.[6] Agda guarantees that run-time irrelevant arguments are not needed for evaluation of the program, they can thus safely be erased by the compiler. For example, we can mark the *n* argument to the tail function as run-time irrelevant:

```
tail' : {@0 n : ℕ} →
  Vec ℕ (suc n) → Vec ℕ n
tail' (x :: xs) = xs
```

In our embedding of PostScript into Agda, we make use of this annotation to ensure that the functions we define do not computationally depend on arguments that are not on the stack and those arguments can hence safely be erased during extraction of PostScript code (see Sect. 3 and Sect. 4).

Generalizable variables. To avoid having to bind implicit arguments in type signatures, we use *generalizable variables*.[7] For example, declaring *n* as a variable allows us to avoid having to bind *n* explicitly in the type of tail:

```
variable @0 n : ℕ
tail'' : Vec ℕ (suc n) → Vec ℕ n
tail'' (x :: xs) = xs
```

Reflected syntax. The reflection API of Agda[8] provides various datatypes that represent the internal syntax of Agda programs. Expressions (of type Term) are represented by a constructor such as con (for constructors), def (for other defined symbols), lam (for lambda expressions), or var (for variables). The constructors con and def are applied to a quoted name (of type Name) and a list of arguments (of type List (Arg Term)). vArg denotes a visible argument, while hArg is used for hidden arguments. For example, the full reflected form of the expression suc zero is con (quote suc) (vArg (con (quote zero) []) :: []).

To make reflected syntax more readable, we use *pattern synonyms*[9] for commonly used pieces of syntax. As a convention, the names of these pattern synonyms start with a backtick followed by the name of the represented Agda construct, for example:

```
pattern `ℕ       = def (quote ℕ) []
pattern `zero    = con (quote zero) []
pattern `suc x   = con (quote suc) (vArg x :: [])
pattern _`+_ x y = def (quote _+_) (vArg x :: vArg y :: [])
```

As a complete example, below is the definition of a function foo (left) and its reflected syntax `foo (right):

```
foo : ℕ → ℕ
foo 0       = zero
foo (suc x) = x + x
```

```
`foo = function
  ( clause [] (vArg `zero :: []) `zero
  :: clause (("x" , vArg `ℕ) :: [])
            (vArg (`suc (var 0)) :: [])
            (var 0 [] `+ var 0 []) :: [] )
```

The reflected syntax of foo (of type Definition) is represented by the constructor function applied to a list of clauses. Each clause (of type Clause) itself is represented by the constructor clause applied to three arguments: i) the telescope, *i.e.* a list of the names of variables and their types; ii) the list of patterns (of type List (Arg Pattern)); and iii) the body of the clause (of type Term). Variables (both in patterns and in terms) are given as de Bruijn indices relative to the telescope of the clause. That is, in the second clause the de Bruijn index 0 refers to the variable *x*. Note that numbers 0, 1, 2,... are expanded into their corresponding zero/suc representations.

The TC monad. Following the approach of *elaborator reflection* introduced by Idris [Christiansen and Brady 2016], Agda exposes many parts of the elaborator to the reflection API, including reduction and normalisation of expressions,

[5] agda.readthedocs.io/en/v2.6.2/language/termination-checking.html
[6] agda.readthedocs.io/en/v2.6.2/language/runtime-irrelevance.html
[7] agda.readthedocs.io/en/v2.6.2/language/generalization-of-declared-variables.html
[8] agda.readthedocs.io/en/v2.6.2/language/reflection.html
[9] agda.readthedocs.io/en/v2.6.2/language/pattern-synonyms.html

through the TC monad. Agda terms can be converted to reflected syntax by using the quoteTC primitive.

Functions of return type Term → TC ⊤ can be marked as a macro. When the elaborator encounters a macro call, it runs the macro and replaces it with the result. A macro can perform arbitrary manipulations on the syntactic structure of Agda expressions as well as information obtained through operations in the TC monad.

3 PostScript and Its Embedding in Agda

PostScript is a document description language, and besides the usual markup, it is possible to define arbitrary computations in it. The language is dynamically typed and stack-based. That is, there is a notion of a global stack that is used for passing values and storing results. All the commands are argumentless operators, and a program is a chain of such commands. For example, consider a function that computes $a^2 + b^2$, where a and b are the top two stack values.

```
dup   % a b b      duplicate top element
mul   % a b*b      multiply top two numbers
exch  % b*b a      excange top two elements
dup   % b*b a a    duplicate top element
mul   % b*b a*a    multiply top two numbers
add   % b*b+a*a    add top two numbers
```

Commands use mnemonic names and typically implement a simple computation or element manipulation on the stack. Recursive function definitions are written as follows:

```
/fib {
  dup 0 eq               % n n==0
  { pop 1 }              % 1
  { dup 1 eq             % n n==1
    { pop 1 }            % 1
    { dup 1 sub fib      % n fib(n-1)
      exch 2 sub fib     % fib(n-1) fib(n-2)
      add                % fib(n-1)+fib(n-2)
    } ifelse
  } ifelse
} def
```

A function is defined with the slash name ('fib' in the above example), followed by a block of commands that are written within braces (the body of the function) followed by the **def** command. Definitions may be used as regular commands, including recursive calls. In the body of the function, we check whether the argument (the top stack element) is zero, in which case we remove it from the stack and put the value one. The same for the case when the argument is one.

Otherwise, we duplicate the argument, subtract one, and make a recursive call. Then we exchange the original argument with the result of the recursive call by running **exch**. We subtract two, make a recursive call and add results of two recursive calls. Conditionals are expressed with two code blocks followed by the **ifelse** command.

In Fig. 1 we draw the results of the fib function (code not shown here) using a PostScript interpreter.

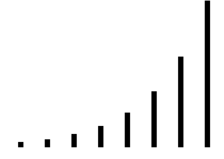

Figure 1. Draw fib.

Assumptions. We consider a small subset of PostScript that is sufficient to express functions on natural numbers. While PostScript has many more commands, types, and drawing primitives, this subset is sufficient to demonstrate the main challenges with verification and extraction. This keeps the complexity of our extractor low, and makes the examples transferable to other stack languages such as Forth.

The main focus of our Agda embedding is to track the number of elements on the stack. On the one hand this helps to entirely avoid stack underflows — an extremely frequent practical problem in stack-based programs. On the other hand, by doing so, we almost immediately run into necessity to use dependent types in the embedded programs. The other points that we want to demonstrate in the embedding are: i) attaching arbitrary properties to function arguments (see for example the index function); ii) guaranteeing function termination; and iii) using runtime-irrelevance annotations to guarantee that extra properties do not have any computational meaning.

3.1 Embedding in Agda

Our PostScript embedding in Agda consists of a type for stacks and a number of basic functions operating on it.

Stack type. We define the type of our stack inductively, and we force the type to carry its length. Per our assumptions, the stack can only store elements of type ℕ.

```
data Stack : @0 ℕ → Set where
  []  : Stack 0
  _#_ : Stack n → ℕ → Stack (suc n)
```

Similarly to vectors, the Stack type has two constructors: [] for stacks of length zero and _#_ for stacks of length 1+n. For example, [] # 1 # 2 # 3 is a stack of type Stack 3. We define _#_ to be left-associative, therefore we do not need any parenthesis. We annotate the index of Stack as computationally irrelevant.

Basic Operations. The basic stack operations are defined as functions from Stack to Stack. The type index makes it possible to capture precisely the effect of each operation.

```
push x s       = s # x      -: ℕ → Stack n → Stack (1 + n)
pop  (s # x)   = s          -: Stack (1 + n) → Stack n
dup  (s # x)   = s # x # x  -: Stack (1 + n) → Stack (2 + n)
exch (s # x # y) = s # y # x -: Stack (2 + n) → Stack (2 + n)
add  (s # x # y) = s # x + y -: Stack (2 + n) → Stack (1 + n)
mul  (s # x # y) = s # x * y -: Stack (2 + n) → Stack (1 + n)
```

In the types of these operations, the length index of Stack ensures that the body of the function respects the specification. If the body of the function returns the stack that does not have the length prescribed by the type, such a function would not typecheck.

Since the size of the stack is an expression that can contain free variables as well as concrete numbers, it is not always possible for Agda to see automatically that two stack sizes are equal. For example, if we require a stack of length $m+n$, but we have a stack of length $n+m$, we cannot blindly use it, as this would not typecheck. To deal with situations like this, we provide an operation subst-stack that cast a stack of length m into the stack of length n, given a (run-time irrelevant) proof that $m \equiv n$.

```
subst-stack : @0(m ≡ n) → Stack m → Stack n
subst-stack refl s = s
```

This operation does not have any run-time behaviour and will be erased by the extractor.

We also define the PostScript command index that makes it possible to access any element of the stack by providing its offset. This can be seen as a more general version of the dup command. The function index requires a proof that the index is within bounds. Also, we are not strictly following the semantics of PostScript, as the index is passed explicitly, rather than taking it from the stack.

```
_!_ : Stack m → (k : ℕ) → @0{T (k < m)} → ℕ
_!_ (s # x) zero            = x
_!_ (s # x) (suc k) {sk<m} = (s ! k) {sk<m}

index : (k : ℕ) → @0{T (k < m)} → Stack m → Stack (1 + m)
index k {k<m} s = s # (s ! k) {k<m}
```

The proof[10] that k is less than m is marked as implicit, which means that Agda will automatically fill in the proof (at least in the simple cases that we have in this paper).

We explicitly forego the definition of conditionals and comparison operators in favour of using pattern-matching functions. Recursion is essential part of Agda, so there is no need to introduce any new operators. In Sect. 3.3 we demonstrate how to add a for-loop to the embedding.

Nothing in this shallow embedding prevents us yet from doing operations that are impossible to express in PostScript, such as duplicating the whole stack or discarding it altogether. Such properties could be enforced by using an (indexed) monad for stack operations, or by working in a quantitative type theory such as Idris 2 [Brady 2021]. In this paper we take a more straightforward approach by rejecting these illegal programs in our extractor.

[10]We use the function T (found in standard library) to convert a boolean predicate _<_ into a type representing its truth value.

3.2 Examples

Let us consider a typical program in the proposed embedding. We express all the operations in terms of base functions defined above. We start with a trivial function that adds one to the top element of the stack.

```
add-1 : Stack (1 + n) → Stack (1 + n)
add-1 s = add (push 1 s)
```

We are required to define the type, which in turn forces us to specify how does the operation change the length of the stack. Stack operators are regular functions, so the chain of applications would be written in reverse, when comparing to the corresponding PostScript program. While this does not affect functionality, it may be aesthetically pleasing to maintain the original order of operators. For this purpose we define an operation _▹_ as $x ▹ f = f\, x$, so we can for example define add-1 instead as add-1 s = s ▹ push 1 ▹ add.

Consider now the example that computes $a^2 + b^2$ where a and b are top two elements of the stack. It can be easier to understand the code if we introduce names for the intermediate states of the stack using let:

```
sqsum : Stack (2 + n) → Stack (1 + n)
sqsum s#a#b = let s#a#b*b     = s#a#b       ▹ dup  ▹ mul
                  s#b*b#a*a = s#a#b*b     ▹ exch ▹ dup ▹ mul
                  s#a*a#b*b = s#b*b#a*a ▹ exch
              in  s#a*a#b*b ▹ add
```

Agda identifiers are chains of almost arbitrary symbols with no spaces, so $s\#a{*}a\#b{*}b$ is a valid variable name.

Pattern Matching. The only way to express conditional in the proposed embedding is by means of pattern matching. Consider the implementation of the Fibonacci function:

```
{-# TERMINATING #-}
fib : Stack (1 + n) → Stack (1 + n)
fib s@(_ # 0)             = s ▹ pop  ▹ push 1
fib s@(_ # 1)             = s ▹ pop  ▹ push 1
fib s@(_ # suc (suc x)) = s ▹ dup  ▹ push 1 ▹ sub ▹ fib
                              ▹ exch ▹ push 2 ▹ sub ▹ fib
                              ▹ add
```

The only unusual thing here is that we match the structure of the stack and the structure of the element simultaneously. For now, it is an exercise to the reader to verify that fib actually implements Fibonacci numbers. Below, we give a formal proof of this fact named ✓.

Note that Agda does not see that the fib function terminates. For now, we add an explicit annotation, but we demonstrate how to deal with this in Sect. 3.3.

Dependent Stack Length. So far, all the specifications in the embedded language did not require dependent types, and could be encoded in languages with a weaker type system such as Haskell or OCaml. However, it quickly becomes

clear that even simple programs in stack languages may expose dependency between the input stack and the size of the output stack. Capturing these cases statically requires dependent types. An example of such a program is a function rep that replicates the x value n times, where x and n are top two stack elements. Here is a possible implementation:

```
{-# TERMINATING #-}
rep : (s : Stack (2 + n)) → Stack ((s ! 0) + n)
rep s@(_ # _ # zero)           = s ▹ pop ▹ pop
rep s#x#m+1@(_ # _ # suc m) =
      let s#x#m    = s#x#m+1  ▹ push 1 ▹ sub
          s#x#m#x = s#x#m     ▹ index 1
          s#x#x#m = s#x#m#x  ▹ exch
      in  subst-stack (+-suc _ _) (rep s#x#x#m)
```

The length of the stack returned by rep is given by the topmost element of the input stack s plus n. Hence the size of the output stack depends on the value of the input stack. In case this argument is zero, we remove two elements from the stack: the argument we were replicating, and the count argument. Otherwise, we decrease the count, copy the argument we are replicating, and put them in the expected position to make the next recursive call. This results in the stack rep *s#x#x#m* of size $(m + (1 + n))$ while the expected size is $(1 + (m + n))$, which are not obviously equal to Agda, hence we apply subst-stack with the proof +-suc from the standard library to convert between these two sizes.

Proving Termination. At this point, we have seen how to write programs in the embedding, express non-trivial properties related to the length of the stack, and verify that a function evaluates to the same results as some other function. One remaining problem is that for some functions, Agda cannot automatically prove termination. For these functions we can either add an annotation as above, or rewrite the definition to make proving termination easier.

The problem with rep is that the recursive call happens on the stack that became one element bigger, yet the top element decreased by one. Therefore, this argument is not strictly structurally smaller, and there are no other decreasing arguments, so the termination checker fails to accept this definition. To fix this is, we can add an extra argument on which the function is structurally decreasing, together with a proof that it is related in some way to the values on the stack. For example, for rep we add an implicit argument k, as well as a proof that the top of the stack is equal to k:

```
rep′ : (s : Stack (2 + n)) → @0 (s ! 0 ≡ k) → Stack ((s ! 0) + n)
rep′ s@(_ # zero)    refl = s ▹ pop ▹ pop
rep′ s@(_ # suc m) refl =
     let s′ = s ▹ push 1 ▹ sub ▹ index 1 ▹ exch
     in  subst-stack (+-suc _ _) (rep′ {k = m} s′ refl)

rep : (s : Stack (2 + n)) → Stack ((s ! 0) + n)
rep s = rep′ s refl
```

As the function is pattern-matching on the top of the stack, and the only value of the _≡_ type is refl, the argument k has to be zero in the first case, and suc m in the second case. This ensures that rep′ is structurally decreasing in k, and the function is accepted by the termination checker.

Showing termination of the fib function fails for the same reason as in case of rep — it is unclear whether any argument decreases when calling fib recursively. Unfortunately, we cannot use the above trick as is. The problem is that after the first recursive call fib *s#x#x-1* we obtain (conceptually) a new stack. To call fib on $x-2$ we first apply exch to the result of the first recursive call (to bring x at the top). However, the termination checker does not see that fib only modified the top element of the stack and did not touch other elements. While we can prove termination of the current version of fib, due to space limitations, we provide an alternative (provably terminating) implementation of fib in Sect. 3.3.

Extrinsic Verification. The nature of dependently-typed systems makes it possible not only to specify functions with intrinsic constraints, such as length of the stack, but also to prove some properties about existing functions as theorems. For example, we prove that sqsum actually implements the sum of squares:

```
sqsum-thm : sqsum (s # k # l) ≡ s # k * k + l * l
sqsum-thm = refl
```

The theorem says that for any s, k and l, application of sqsum to s appended with k and l equals to s appended with $k^2 + l^2$. Luckily, from the way we constructed the basic operations, this fact is obvious to Agda. So the proof is simply the reflexivity constructor.

On the other hand, proving that fib matches a simpler specification that we call fspec requires a bit more work.

```
fspec :                       ✓ : (s : Stack n) (x : ℕ)
  ℕ → ℕ                          → fib (s # x) ≡ s # fspec x
fspec 0 = 1                   ✓ s 0 = refl
fspec 1 = 1                   ✓ s 1 = refl
fspec (suc (suc x)) =         ✓ s (suc (suc x)) rewrite
     fspec (suc x)               ✓ (s # suc (suc x)) (suc x) |
  + fspec x                      ✓ (s # fspec (suc x)) x = refl
```

This is an inductive proof where we consider two base cases, and the step case. In the latter we refer to the theorem with a structurally smaller arguments, and after rewriting such cases, the statement becomes obvious.

3.3 For Loop

The final part of our embedding is the for-loop construct. Not only this is often found in real PostScript documents, it also helps to avoid the problem with proving termination. The difficulty with encoding the for-loop behaviour lies in

its potential ability to arbitrarily modify stack at every iteration. While there is no technical problem to encode[11] such a behaviour in Agda, it would be quite inconvenient to work with. Every time one needs to ensure that a stack returned by a for-loop contains enough elements, a potentially complex proof has to be given. We make our life easier by working with a simpler version of the for loop that assumes the same stack size at each iteration, which is sufficient for our examples. Concretely, the boundaries of the loop are given by two numbers s and e, where the loop iterates through indices $s, 1 + s, \ldots, e$.

We define for-loop as a function of two arguments: the body of the for-loop given by a function and the initial stack.

```
for : (Stack (1 + n) → Stack n) → Stack (2 + n) → Stack n
for {n} f (st # s # e) = if s ≤ e then loop (e - s) st else st
  where loop : ℕ → Stack n → Stack n
        loop zero    st = st ▹ push s ▹ f
        loop (suc i) st = st ▹ loop i ▹ push (suc i + s) ▹ f
```

The initial stack contains 2 loop boundary elements and n other elements. The implementation of for computes the number of iterations i and unrolls the loop that many times, each time pushing the current value of the loop variable to the top of the stack. In the end, it finishes with a stack with n elements. If the lower boundary is already above the upper boundary initially, it removes both of them and returns immediately.

Now we are ready to define our fib example using for:

```
fib-for : Stack (1 + n) → Stack (1 + n)
fib-for s@(_ # x) =
  s ▹ push 1 ▹ exch ▹ push 1 ▹ exch ▹ push 1 ▹ exch
    ▹ for (λ s → s ▹ pop ▹ exch ▹ index 1 ▹ add) ▹ pop
```

Our initial stack contains the function argument x at the top. We modify the stack by inserting 1 and 1 (inital fibonacci seeds) and 1 (the lower bound for the for loop) before x. In the function body, we remove the iteration value, and modify $\langle a, b\rangle$ into $\langle b, a+b\rangle$. Note that termination of fib-for is derived automatically.

[11] If one wants to mimic the actual for-loop found in PostScript, the type of the operation would be:

```
for : {f : ∀{m} → Stack (1 + m) → ℕ}
    → (loop : ∀ {m} (s : Stack (1 + m)) → Stack (f s))
    → ∀ {m} (s : Stack (3 + m)) → Stack (#it loop s)
```

As it can be seen, we have to explain how the size of the stack changes at each iteration, which is given by a function f that determines the size of the stack after each iteration, as this can depend on the values found on the stack. Unfortunately, the size of the stack after the final iteration cannot be computed upfront. We would have to run the loop to get the size — this is done by #it. However, working with such encoding is very inconvenient. Even simplest analysis of determining whether for-loop returns a non-empty stack turns into painful proving exercise. Therefore, it is more practical to introduce well-behaved variants of the for-loop where we have strong guarantees about the final size of the stack. All such variants can be seen as special cases of the generic for loop.

We now consider a more realistic PostScript example that generates an image of Sierpinski fractal. The structure of the code consists of a doubly-nested for loop that draws a dot at each coordinate (i, j) where the bit-wise 'and' of i and j is zero. For this example we assume that a drawing function and bit-wise 'and' are already defined in PostScript, so we postulate them in Agda. This means that we only provide a type signature of the functions, but not the implementation. We implement conditional drawing via the helper function draw-if.

```
postulate draw-circ-xy : Stack (2 + n) → Stack n
          bit-and : Stack (2 + n) → Stack (1 + n)

draw-if : Stack (3 + n) → Stack (2 + n)
draw-if s@(_ # 0) = s ▹ pop ▹ index 1 ▹ index 1
                      ▹ draw-circ-xy
draw-if s          = s ▹ pop
```

The main function sets the boundaries for both for-loops, applies bit-and to i and j, and calls the drawing function, ensuring that no extra arguments are left on the stack.

```
sierpinski : Stack (1 + n) → Stack n
sierpinski s =
  s ▹ push 0 ▹ index 1
    ▹ for (λ s → s ▹ push 0 ▹ index 2
                   ▹ for (λ s → s ▹ index 1 ▹ index 1
                                  ▹ bit-and ▹ draw-if ▹ pop)
                   ▹ pop)
    ▹ pop
```

When implementing algorithms like this one manually, it is easy to forget to remove or copy an element in the body of the for-loop. The strict stack size discipline that we have in Agda helps to avoid these errors.

4 Extraction

In this section, we show a concrete example of an extractor implemented using reflection in Agda.

Assumptions. Our extractor serves a dual purpose. On the one hand, we traverse Agda terms and map basic stack operations such as dup and add to their PostScript counterparts. On the other hand, the extractor determines which terms are valid in the presented shallow embedding: these are the terms that are accepted by our extractor. Our criteria of acceptable embeddings are as follows:

- The function acts on a single stack argument and returns stack. The function can accept arbitrary number of additional arguments (for verification purposes) as long as these arguments are computationally irrelevant. This is checked by the function extract-type.
- Within the function, the stack is never duplicated, discarded or modified by any means but embedded stack operations. This is ensured by the functions extract-term and stack-ok.

- Conditionals for stack elements are implemented using pattern-matching. The extractor needs to translate these patterns to conditional statements. This is done by extract-pattern and extract-clauses.

4.1 Target Syntax

In the end, the extractor outputs the PostScript syntax as a plain string. However, it is useful to work with a basic abstract syntax representation as an intermediate stage:

```
data PsCmd : Set where
  Pop Dup Exch Add Mul Eq Ge And : PsCmd
  Push Index : ℕ → PsCmd
  FunCall : String → PsCmd
  For     : List PsCmd → PsCmd
  IfElse  : List PsCmd → List PsCmd → PsCmd
  FunDef  : String → List PsCmd → PsCmd
```

We implement a basic pretty-printer print-ps : List PsCmd → String whose definition we omit here.

4.2 The Extraction Monad

We make use of a monad for extraction to keep track of the current state of functions that still need to be extracted, and for propagating errors. The monad combines the built-in TC monad, extraction state and a possibility to chose between values and errors. ExtractM gives rise to do-notation[12]. The fail function aborts the extraction process. Two operations for managing the queue of functions to be extracted: mark-todo adds a function name to the queue, while get-next-todo returns the next function that has been marked for extraction. Finally, the monad provides two operations for getting normalized types and definitions of a given symbol. This can be used for example for inlining Agda functions that cannot be translated to PostScript, or for applying domain-specific optimizations through the use of rewrite rules (Sect. 5).

```
record ExtractM (X : Set) : Set
fail                : String → ExtractM X
mark-todo           : Name → ExtractM ⊤
get-next-todo       : ExtractM (Maybe Name)
get-normalised-type : Name → ExtractM Type
get-normalised-def  : Name → ExtractM Definition
```

4.3 The Extractor

The extractor itself consists of four functions that traverse the different parts of the reflected Agda syntax and translate it to PostScript commands. In the remainder of this section, we explain the implementation of these functions in detail.

Extracting terms. extract-term traverses an Agda term and translates it to a list of PostScript commands. For example, add (push 1 *s*) is translated to Push 1 :: Add :: []. It takes an additional argument of type Pattern to check that the stack used in the expression (in this case *s*) is identical to the input stack. In this way it ensures that we do not manipulate the stack in arbitrary ways, but only through the primitive stack operations of PostScript. The implementation of extract-term uses a helper function go to traverse the reflected Agda syntax, collecting the generated PostScript commands in an accumulator. Note that we defined a number of pattern synonyms such as ‘pop, ‘dup, *etc.*

```
-: Term → Pattern → ExtractM (List PsCmd)
extract-term v stackp = go v []
  where
  go : Term → List PsCmd → ExtractM (List PsCmd)
```

The cases for basic instructions are as follows:

```
  go (‘pop  x) acc = go x (Pop  :: acc)
  go (‘dup  x) acc = go x (Dup  :: acc)
  go (‘exch x) acc = go x (Exch :: acc)
  go (‘add  x) acc = go x (Add  :: acc)
  go (‘mul  x) acc = go x (Mul  :: acc)
```

For the commands push and index, the extractor currently only allows natural number literals 0, 1, 2, ... as the argument. For any other argument the extraction is aborted by calling the fail function.

```
  go (‘push (‘num n) x) acc = go x (Push n :: acc)
  go (‘push k _) acc = fail ("push non-literal: " <>ₜ k)
  go (‘index (‘num n) x) acc = go x (Index n :: acc)
  go (‘index k _) acc = fail ("index non-literal: " <>ₜ k)
```

The function subst-stack is only needed to satisfy the Agda typechecker, but does not have any run-time behaviour. Hence it is erased during extraction.

```
  go (‘subst-stack x) acc = go x acc
```

To extract a for loop, we first check that the body of the loop is a lambda term. If that is the case, we extract the body *b*, using the stack pattern (var 0) that refers to the stack variable bound by the lambda. After the body of the loop has been extracted, we construct the For node and continue extraction with the expression for the initial stack *x*.

```
  go (‘for (vLam _ b) x) acc = do
    proc ← extract-term b (var 0)
    go x (For proc :: acc)
  go (‘for b _) acc = fail ("invalid for body: " <>ₜ b)
```

When it reaches a defined function that is not in the set of base functions, the extraction proceeds in three steps. First, it adds the function to the queue for later extraction using mark-todo. Next, it gets the type of the function and calls extract-type to determine the index of its principal argument. Finally, it looks up the corresponding argument in the argument list and continues extraction with that argument.

```
  go (def f args@(_ :: _)) acc = do
    mark-todo f
```

[12] agda.readthedocs.io/en/v2.6.2/language/syntactic-sugar.html#do-notation

```
    ty ← get-normalised-type f
    n  ← extract-type ty
    a  ← lookup-arg args n
    go a (FunCall (prettyName f) :: acc)
```

After traversing through the stack operations, we reach the stack itself. Here we check that the stack that is used is the same as the input stack, which is done by stack-ok (explained below). If the check succeeds, we return the list of commands collected in *acc*.

```
  go v acc = do
    b ← stack-ok stackp v
    if b then (return acc)
      else (fail ("stack mismatch: "
           <> showPattern stackp <> " and " <>t v))
```

The function stack-ok ensures that when we use the stack (of type Term), it is identical to the stack that we got as the input tothe function (of type Pattern). In addition to the cases below, there are a few other cases for dealing with natural number literals 0, 1, 2, ... (not shown here).

```
-: Pattern → Term → ExtractM Bool
stack-ok p@(p₁ ‘# p₂) t@(t₁ ‘# t₂) = do
  ok₁ ← stack-ok p₁ t₁
  ok₂ ← stack-ok p₂ t₂
  return (ok₁ ∧ ok₂)
stack-ok (var x) (var y []) = return (x ℕ.≡ᵇ y)
stack-ok ‘zero    ‘zero     = return true
stack-ok (‘suc x) (‘suc y)  = stack-ok x y
stack-ok p t = return false
```

Extracting types. The function extract-type defines what Agda types are valid for functions in the embedding. It traverses an Agda type and checks that it takes one principal argument of type Stack and returns a value of type Stack. In addition, it checks that all non-principal arguments to the function are marked as runtime-irrelevant and can thus safely be erased during extraction. If these checks succeed, it returns the position of the principal argument.

```
-: Type → ExtractM ℕ
extract-type x = go x false 0
  where
  go : Type → (st-arg : Bool) → (idx : ℕ) → ExtractM ℕ
  go (Π[ s : vArg (‘Stack n) ] ty) false i = go ty true i
  go (Π[ s : erasedArg _ ] ty) b i =
    go ty b (if b then i else 1 + i)
  go (‘Stack n) true i = return i
  go t _ _ = fail ("invalid type: " <>t t)
```

Extracting clauses. extract-clauses takes as input the clauses of a function definition and the position of the principal argument (as computed by extract-type) and translates the clauses to a list of PostScript commands. For example, consider the function non-zero:

```
non-zero : Stack (1 + n) → Stack (1 + n)
non-zero s@(_ # 0) = s
non-zero s@(_ # _) = s ▹ pop ▹ push 1
```

The clauses of non-zero are translated to a conditional expression in PostScript that checks whether the top element is zero:

```
0 index 0 eq { } { pop 1 } ifelse
```

The two helper functions extract-natp and extract-stackp extract a boolean condition from a given Agda pattern. First, extract-natp compiles a pattern of type ℕ to just a condition on the given position on the stack, or nothing if the pattern matches unconditionally. There are three cases:

- A variable pattern *n* matches any input, so nothing is returned.
- A closed pattern suc (suc (... zero)) only matches the single value equal to the number of successors, so we return an equality check.
- A successor pattern suc (suc (... *n*)) matches any value greater or equal to the number of successors, so we return an inequality check.

In the implementation below, the argument *c* keeps track of the number of successors encountered so far.

```
-: ℕ → Pattern → ExtractM (Maybe (List PsCmd))
extract-natp hd-idx p = go p 0
  where
  mk-cmp : PsCmd → ℕ → List PsCmd
  mk-cmp cmp n = Index hd-idx :: Push n :: cmp :: []

  go : Pattern → ℕ → ExtractM (Maybe (List PsCmd))
  go (var _)   0 = return nothing
  go (var _)   c = return (just (mk-cmp Ge c))
  go ‘zero     c = return (just (mk-cmp Eq c))
  go (‘suc p)  c = go p (1 + c)
  go (‘num n) c = return (just (mk-cmp Eq (c + n)))
  go p         c = fail ("not a nat: " <> showPattern p)
```

Second, the function extract-stackp compiles a pattern of type Stack to just the condition as a list of PostScript commands, or nothing in case the pattern is guaranteed to match. There are two cases:

- A variable pattern *s* matches any input, so nothing is returned.
- A stack pattern *ps* # *p* matches if the top of the stack matches *p* and the remainder matches *ps*. In case both patterns require non-trivial conditions, we combine both using the And instruction.

```
-: ℕ → Pattern → ExtractM (Maybe (List PsCmd))
extract-stackp hd-idx (var x)    = return nothing
extract-stackp hd-idx (ps ‘# p)  = do
  ml₁ ← extract-natp hd-idx p
```

```
  ml₂ ← extract-stackp (offset ml₁ + hd-idx) ps
  return (combine ml₁ ml₂)
  where
    offset nothing = 1
    offset (just _)  = 2
    combine nothing ml₂        = ml₂
    combine ml₁        nothing = ml₁
    combine (just l₁) (just l₂) = just (l₁ ++ l₂ ++ [ And ])
extract-stackp _        p =
  fail ("invalid stack pattern" <> showPattern p)
```

We are now ready to implement extraction of function clauses. The extraction of a clause clause _ *ps t* with patterns *p* and right-hand side *t* proceeds by first looking up the pattern *stackp* corresponding to the principal argument, compiling this pattern to a condition using extract-stackp, and (if the condition is non-trivial) recursively extracting the remaining clauses. The final result uses IfElse to select the right clause.

When compiling the final clause, we skip compilation of the pattern. This is a correct optimization because Agda enforces completeness of definitions by pattern matching, so if the final case is reached it is guaranteed to match.

```
-: Clauses → ℕ → ExtractM (List PsCmd)
extract-clauses (clause _ ps t :: []) i = do
  stackp ← lookup-arg ps i
  extract-term t stackp
extract-clauses (clause _ ps t :: ts) i = do
  stackp ← lookup-arg ps i
  just l  ← extract-stackp 0 stackp
    where nothing → extract-term t stackp
  t ← extract-term t stackp
  ts ← extract-clauses ts i
  return (l ++ [ IfElse t ts ])
extract-clauses [] i = return []
```

Extracting definitions. Finally, extract-def takes as input a (reflected) name of an Agda function, gets its type and definition, and calls extract-type and extract-clauses to translate it to a list of PostScript commands.

```
-: Name → ExtractM (List PsCmd)
extract-def f = do
  ty   ← get-normalised-type f
  function cs ← get-normalised-def f
    where _ → return []
  i ← extract-type ty
  b ← extract-clauses cs i
  return [ FunDef (prettyName f) b ]
```

Whole program extraction. To run the extractor on a complete Agda program, we need to run it on the main function and all its (recursive) dependencies. This is implemented by the function extract-defs, which makes use of get-next-todo of the extraction monad to extract all function definitions one by one. Since any Agda program has a finite number of definitions and each definition is only processed once, this function is terminating. However, the Agda termination checker does not detect this, so we mark it as terminating manually using a pragma.

```
{-# TERMINATING #-}
extract-defs : ExtractM (List PsCmd)
extract-defs = do
  just f ← get-next-todo
    where nothing → return []
  xs ← extract-def f
  ys ← extract-defs
  return (xs ++ ys)
```

We define a macro extract as the main entry point of the extractor. This macro takes as inputs the name of the main function and a list of functions that should not be inlined (see the next section for more details on inlining). The implementation of the macro (not shown here) runs extract-defs on the initial state. If extraction succeeds, it replaces the call to the macro by the pretty-printed result, and otherwise throws an error.

```
macro extract : Name → Names → Term → TC ⊤
```

We provide a default list base of functions for which to avoid inlining, which can be further tailored to the extraction of a specific program.

Testing the extractor. Thanks to the theorem-proving capabilities of Agda, we can embed test cases for the extractor as equality proofs. These test cases are run automatically during type checking, so if a change to the extractor causes one of them to fail it will not go unnoticed.

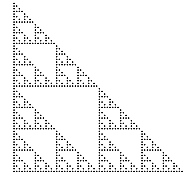

Figure 2. Draw sierpinski.

As an example, we test that add-1 is extracted correctly:

```
test-add1 : extract add-1 base
            ≡ "/add-1 {\n  1 add\n} def\n"
test-add1 = refl
```

We can test the output of the extractor on the other examples from the previous section in a similar fashion. Finally, we can feed generated programs into PostScript interpreters and obtain outputs such as one at Fig. 2.

5 Partial Evaluation

Working with a shallow embedding brings us an important benefit: we can use the existing evaluator of Agda to partially evaluate programs prior to extraction. In this section, we give a couple of examples of how this is useful. We also demonstrate how to extend Agda's evaluator with domain-specific optimizations through the use of *rewrite rules.*

Using Agda functions as macros. By reducing Agda expressions prior to extraction, we may use any host language constructs that are not present in the embedding, as long as they are eliminated prior to extraction. For example, we can use of the Agda function applyN to apply a certain postscript operator n times:

```
applyN : ℕ → (X → X) → X → X
applyN zero    f x = x
applyN (suc n) f x = f (applyN n f x)

pow32 : Stack (1 + n) → Stack (1 + n)
pow32 s = applyN 5 (λ s → s ▹ dup ▹ mul) s
```

The function applyN is a polymorphic and higher-order function, so it falls well outside the fragment of Agda that our extractor can deal with. Nevertheless, the extractor can inline the definition of applyN: running extract pow32 base produces the following code:

```
/pow32 {
  dup mul dup mul dup mul dup mul dup mul
} def
```

In essence, this allows us to write macros using arbitrary Agda functions, as long as the end result falls within the fragment that the extractor knows how to deal with.

Partial evaluation of primitive operators. In addition to inlining external functions, the extractor can also simplify expressions that involve basic operations such as push and pop. To achieve this, we pass an empty list as the second argument to the extract macro (which is the list of functions that should not be inlined). For example, it can eliminate values that are first pushed and then popped again without being used:

```
push-pop : Stack n → Stack n
push-pop s = s ▹ push 42 ▹ pop
```

```
/push-pop {
} def
```

Using the same technique with the for for, we can automatically unroll loops with constant boundaries for free.

Domain-specific optimizations as rewrite rules. One common way to define domain-specific compiler optimizations is through the specification of *rewrite rules* that rewrite terms matching a given pattern to an equivalent form that is either more efficient or reveals further optimization opportunities. By giving a shallow embedding of our target language in Agda, we have the opportunity to define *verified* rewrite rules, providing a proof that the left- and right-hand side of the rewrite rule are equivalent. To achieve this, we could define our own representation of verified rewrite rules and integrate them into the extractor. However, we can avoid the effort of doing so since Agda already has a built-in concept of rewrite rules [Cockx 2019].

As an example, we prove that pushing and then adding two numbers in sequence is equivalent to pushing and adding the sum of these numbers.

```
add-add-join : (s : Stack (1 + n)) (k l : ℕ)
  → s ▹ push k ▹ add ▹ push l ▹ add ≡ s ▹ push (k + l) ▹ add
add-add-join (s # x) k l = cong (s #_) (+-assoc x k l)
```

Next, we register this equality as a rule to be applied automatically during evaluation by using a REWRITE pragma:

```
{-# REWRITE add-add-join #-}
```

From now on the rule is applied automatically by the extractor whenever possible:

```
add-numbers : Stack (1 + n) → Stack (1 + n)
add-numbers s = s ▹ push 1 ▹ add ▹ push 2 ▹ add
                  ▹ push 4 ▹ add ▹ push 2 ▹ add
```

Running extract add-numbers [] produces this code:

```
/add-numbers
{ 9 add
} def
```

Implementation details. Partial evaluation in Agda is achieved by normalising, *i.e.* by applying reduction rules to (sub)terms until they turn into values or neutral terms. Agda's reflection API offers a function normalise for this purpose. However, using this function we ran into two problems:

- The normalise function only works on terms and not on entire function definitions. Hence we manually traverse the function definition and call normalise on the body of each individual clause. During the implementation of this traversal, we were faced with the challenge of reconstructing the right typing context for each clause. Agda constructs this context internally during elaboration of the clauses, but the reflection API did not provide access to it. To solve this problem we extended the reflection API to provide it for us.[13]
- The functionality to selectively normalise certain functions while leaving others intact was not previously available in Agda. We added dontReduceDefs and onlyReduceDefs to the reflection API.[14]

These two changes to Agda were motivated by our goal to implement custom extractors through reflection, but they are generally useful for users of the reflection API. Both changes have been released as part of Agda 2.6.2.

6 Related Work

There is a large body of work on metaprogramming facilities in various programming languages. Herzeel et al. [2008] track the origins of metaprogramming to Smith's work on reflection in Lisp [Smith 1984]. Some prominent metaprogramming systems include MetaOCaml [Kiselyov 2014], MetaML [Taha and Sheard 1997], reFlect [Grundy et al. 2006], Template Haskell [Sheard and Peyton Jones 2002], Racket

[13] See github.com/agda/agda/pull/4722.

[14] See github.com/agda/agda/pull/4978.

[Flatt and PLT 2010], and various other Lisp/Scheme dialects. However, these systems typically do not support dependent types, so they are not well suited for our goal of statically enforcing correctness of embedded programs.

Defining deep embeddings with static guarantees are a common application of dependent types [Allais et al. 2018; Altenkirch and Reus 1999; Chapman 2009; Danielsson 2007; McBride 2010]. These embeddings usually also define semantics of the embedded language and therefore allow us to reason about the correctness of program transformations and optimisations. While the this is impressive in theory, the resulting encodings are difficult to use in practice. In this paper we instead aim for a more lightweight approach. Svenningsson and Axelsson [2013] propose to solve this problem with a combination of deep and shallow embeddings, by using a small deep embedding and leveraging type classes in Haskell to define the rest of the language on top of that. However, so far this idea has not yet been applied to dependently typed embedded languages.

The Coq proof assistant is equipped with extraction capabilities [Letouzey 2003, 2008], which extracts functional code from Coq proofs (or programs). The default target language is OCaml, but a few other options were added recently. Likewise, Agda itself has a mechanism for defining custom backends, of which the GHC backend is the most prominent. Other proof assistants provide similar extraction tools as well. The main difference from our approach in this paper is that these extractors are written as plugins to the proof assistant, while we implement our extractors directly in the proof assistant itself. As a consequence, our extractors and programs can (in principle) communicate with each other. In addition, as they are just Agda programs, they can be reflected themselves and their structure can be leveraged.

Several dependently-typed languages are equipped with metaprogramming capabilities: Idris [Christiansen and Brady 2016], Lean [Ebner et al. 2017], Coq [Anand et al. 2018], and Agda [van der Walt and Swierstra 2013]. All of these implement a similar API as described in this paper, and hence could be used to implement our approach — one simply needs to adjust extraction to the particular API. Chang et al. [2019] introduce the Turnstile+ framework for building dependently typed embedded DSLs and shares the ideas advocated in this paper, suggesting that our approach could work there as well. Sozeau et al. [2019] use MetaCoq to formally verify the core type system of Coq. This combines nicely with our approach, as we could use the verified core language as a basis to verify our custom extractors. Annenkov et al. [2020] use MetaCoq to implement a DSL combining deep and shallow approaches, in a way that is quite similar to our own. While they are able to formally reason about preservation of semantics (which we cannot do yet), it is unclear whether their approach scales to dependently-typed embedded languages.

7 Conclusions and Future Work

In this paper we investigate the idea of developing embedded programs hand-in-hand with custom code generators for them. We solve the well-known conundrum of choosing between deep and shallow embedding by leveraging the power of reflection. This allows us to enjoy simplicity of writing programs in shallow embedding, while keeping the ability to translate them into the original language.

We apply this idea in the context of dependently-typed language Agda. We demonstrate the use of dependent types to enforce static properties of the embedded programs. Concretely, we have demonstrated the approach by implementing an extractor for a fragment of the PostScript language.

The main advantage of our approach is twofold: first, it allows you to implement all these things — as well as the embedded programs themselves — *side by side in a single language*, simplifying the development. And secondly, it allows you to reuse the typechecker and evaluator of the host language, saving you a lot of work. While we showcase a single example, the approach has been applied to several other languages: Kaleidoscope, a minimalist imperative language; Single assignment C, a high-performance array language; and APL (Array Programming Language), another array language with heavily overloaded syntax [Sinkarovs and Cockx 2021].

We believe that the reader is now empowered to apply the proposed technique to verify code in any language.

Future work. While our approach is flexible, it is not a magic bullet. Right now, we cannot yet guarantee that the extracted code preservers the semantics of the original implementation. While we rarely see fully-verified compiler backends in the real world, our approach is very close to enabling this. In future work, we would like to give a formal semantics of the reflected language and the proof that reflected programs respect it. While this is non-trivial, a system like Agda could do it in principle.

Acknowledgments

We would like to thank reviewers for their constructive suggestions. This work is supported by the Engineering and Physical Sciences Research Council through the grant EP/N028201/1, and the Dutch Research Council through the grant VI.Veni.202.216.

References

Adobe Systems Incorporated. 1999. PostScript Language Reference Third Edition.

Agda Development Team. 2021. *Agda 2.6.2 documentation.* https://agda.readthedocs.io/en/v2.6.2/ Accessed [2021/07/10].

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Proc. ACM Program. Lang.* 2, ICFP, Article 90 (July 2018), 30 pages. https://doi.org/10.1145/3236785

Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic (CSL '99)*. Springer-Verlag, Berlin, Heidelberg, 453–468.

Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 20–39.

Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: A Smart Contract Certification Framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 215–228. https://doi.org/10.1145/3372885.3373829

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2007. Finally Tagless, Partially Evaluated. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–238.

Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. 2019. Dependent Type Systems as Macros. *Proc. ACM Program. Lang.* 4, POPL, Article 3 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371071

James Chapman. 2009. Type Theory Should Eat Itself. *Electronic Notes in Theoretical Computer Science* 228 (2009), 21–36. https://doi.org/10.1016/j.entcs.2008.12.114 Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).

David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 284–297. https://doi.org/10.1145/2951913.2951932

Jesper Cockx. 2019. Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules. In *25th International Conference on Types for Proofs and Programs, TYPES 2019, June 11-14, 2019, Oslo, Norway (LIPIcs, Vol. 175)*, Marc Bezem and Assia Mahboubi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.TYPES.2019.2

The Coq Development Team. 2021. *The Coq proof assistant reference manual.* https://coq.github.io/doc/v8.13/refman/ Version 8.13.2.

Nils Anders Danielsson. 2007. A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family. In *Types for Proofs and Programs*, Thorsten Altenkirch and Conor McBride (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 93–109.

Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 34 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110278

Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

Jim Grundy, Thomas F. Melham, and John W. O'Leary. 2006. A reflective functional language for hardware design and theorem proving. *J. Funct. Program.* 16, 2 (2006), 157–196. https://doi.org/10.1017/S0956796805005757

Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt. 2008. Reflection for the Masses. In *Self-Sustaining Systems*, Robert Hirschfeld and Kim Rose (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 87–122.

Oleg Kiselyov. 2014. The Design and Implementation of MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Pierre Letouzey. 2003. A New Extraction for Coq. In *Types for Proofs and Programs*, Herman Geuvers and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 200–219.

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 359–369.

Per Martin-Löf. 1998. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*, Giovanni Sambin and Jan M. Smith (Eds.). Oxford Logic Guides, Vol. 36. Oxford University Press, 127–172.

Conor McBride. 2010. Outrageous but Meaningful Coincidences: Dependent Type-Safe Syntax and Evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming* (Baltimore, Maryland, USA) *(WGP '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1863495.1863497

Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Emir Pasalic, Walid Taha, and Tim Sheard. 2002. Tagless staged interpreters for typed languages. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional Programming (ICFP 2002)*. 218–229. https://doi.org/10.1145/581478.581499

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 Haskell Workshop, Pittsburgh* (proceedings of the 2002 haskell workshop, pittsburgh ed.). 1–16.

Artjoms Sinkarovs and Jesper Cockx. 2021. Choosing is Losing: How to combine the benefits of shallow and deep embeddings through reflection. *CoRR* abs/2105.10819 (2021). arXiv:2105.10819 https://arxiv.org/abs/2105.10819

Brian Cantwell Smith. 1984. Reflection and Semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Salt Lake City, Utah, USA) *(POPL '84)*. Association for Computing Machinery, New York, NY, USA, 23–35. https://doi.org/10.1145/800017.800513

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 8 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371076

Josef Svenningsson and Emil Axelsson. 2013. Combining Deep and Shallow Embedding for EDSL. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–36.

Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. *SIGPLAN Not.* 32, 12 (Dec. 1997), 203–217. https://doi.org/10.1145/258994.259019

Paul van der Walt and Wouter Swierstra. 2013. Engineering Proof by Reflection in Agda. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.