Proving Compiler Correctness with Dependent Types

João Paulo Pizani Flor <j.p.pizaniflor@students.uu.nl>
Wout Elsinghorst <w.l.elsinghorst@students.uu.nl>

Tuesday 22nd April, 2014

1 Introduction

In this report we describe our work for the final project of the master course ”Theory of Programming and Types” at Utrecht University. Our research involves the question of compiler correctness, i.e, giving a *specification* for a language (a semantics), and proving that a compiler for that language respects the given semantics.

Concretely, our notion of correctness depends on two semantics, respectively for the source and object languages of the compiler. We define a denotational semantics (`eval`) for the source language, as well as an operational semantics (`exec`) for the target machine. Correctness states that evaluation is equal to compilation composed with execution.

More specifically, we were interested in having *machine-checked* proofs of correctness, i.e, proofs written in the language of an interactive proof assistant. Some initiatives in this direction are already being taken, most famously CompCert, a formally verified compiler for the C language, which had its proof written in the *Coq* proof assistant.

Dependent types allow us to embed, as indices to the language definitions and in the type of the “compile” function, constraints which make these definitions correct.

The rest of the report is organized as follows: on section 2 we present the related papers which served as the basis for our research, and state precisely which were our contributions. In section 3 we present the definitions for our *source* and *object* languages and give the implementation of the compiler itself. In section 4 we define a notion of compiler correctness and prove that our compiler is correct according to this definition. Section *sec:lifting* introduces a smarter version of the object language, in which sharing is captured, together with a compiler which produces code in this new language. We then proceed to describe how, given a correctness proof for a “naïve” compiler, a proof of correctness for the smarter version can be obtained.

Acknowledgments: We are pleased to thank Wouter Swierstra for his help and valuable insights.

2 Related work/Goals

On researching the topic of compiler correctness in the context of dependent types, we first encountered the paper “A type-correct, stack-safe, provably correct expression compiler in Epigram” [2]. In this paper, a *very simple* (but *typed*) expression language is presented, along with a *typed bytecode* definition. The correctness of this compiler is then formulated and proven.

Both language definitions, the semantics for each of them, the compiler and its correctness proof were all written in Epigram, a dependently-typed programming language. Therefore, even though the authors admit that the paper carries no real novelty value related to compiler construction, we still felt this paper could serve well as the “basis” for our project, as they treat compilation of *typed source* to *typed bytecode*, and also use a dependently-typed implementation language.

So our work started with trying to extend the source language from [2], and add constructs to it that made it more powerful. Then we came across the paper “Proving Correctness of Compilers using Structured Graphs”[1]. In this paper, the author discusses the issue of *sharing* of bytecode, that is, that some high-level language constructs (typically control flow related constructs, such as exception handling of conditional branching) map into low-level code which has *shared blocks of code among different execution paths*.

One way to compile these control flow constructs would be to extend the bytecode language with explicit jumps and labels, a solution which is often taken in “real-world” compilers, but which makes analyzing bytecode and proving compiler correctness much more complex. A simpler way to handle these situations is to just *replicate* the shared code when compiling constructs which generate different execution paths.

A compiler which works by using a jump-and-label-free bytecode and replicates shared code is easier to analyze, but the practically desirable behaviour is, off course, to represent the sharing. The main contribution of [1] is a systematic way in which a proof of correctness for a “naïve” (code-duplicating) compiler can be used to construct the correctness proof for a “sharing optimized” version of that compiler; a rather elegant approach.

The overall goal of our project was, therefore, to integrate the solutions given both in [2] and [1]. To achieve this goal we needed to make some adjustments to the definitions and proofs of the reference papers. These adjustments are what we perceive as our main contributions with this project:

- The solution in [1] used Haskell as implementation language (along with some proofs given in Coq).
  - We needed, therefore, to adapt definitions such as fixed point operators and generic maps/folds to the *total* setting of Agda.
- The example bytecode language used in [1] to illustrate the method is *untyped*.
  - To make the proof derivation scheme work for *typed* bytecode, we needed to also adapt most of the generic data structures presented to become *indexed*.
- The example control flow construct used in [1] to introduce sharing (exceptions) was also not immediately applicable to being modeled in a total setting.
  - Specifically, the way in which it was implemented (by using Higher-Order Abstract Syntax) clashed with Agda’s requirement of *strict positivity*[1] Exceptions required the `Stack` datatype to contain handlers of the form $Stack \to Stack$, violating the positivity requirement.
  - We chose a simpler sharing-inducing extension to the language of [2] (sequencing and conditional branching). This extension is explained in more detail in section 3

3 Source and Target Language

Our source language (`Src`) is based upon, and therefore very similar, to the one defined in [2]. It is a typed expression language, in which the types are naturals and booleans. There are no binding or application constructs in this language, therefore no arrow type constructor. First, we show the type language for Src:

When naming constructs of the languages which are our ”object of study”, we give subscripts to distinguish them from similarly named definitions in Agda (the *metalanguage*).

```
data Tyₛ : Set where
  ℕₛ : Tyₛ
  𝔹ₛ : Tyₛ
```

Figure 1: Types for our source language.

```
data Src : (t : Tyₛ) → (z : Sizeₛ) → Set where
  vₛ        : ∀ {t} → (v : [[ t ]]) → Src t 1
  _+ₛ_      : (e₁ e₂ : Src ℕₛ 1) → Src ℕₛ 1
  ifₛ_thenₛ_elseₛ_ : ∀ {t n} → (c : Src 𝔹ₛ 1)
    → (eₜ eₑ : Src t (suc n)) → Src t (suc n)
  _⟫ₛ_      : ∀ {t m n} → Src t (suc m) → Src t (suc n) → Src t (suc n + suc m)
```

Figure 2: The `Src` datatype definition.

In the case of the source language, the names are subscripted by a small-case "s".

The definition of `Src` makes clear what is the main difference between our source language and the used defined in [2]: we have an additional **sequencing** construct. Even though the datatype definition is increased by only one constructor, this change has several subtleties:

- All subexpressions in a sequence are required to be of the same base type
- No arithmetic or boolean operators over sequences
  - This is enforced by making expressions have a *size*

As important as defining the constraints on well-formed expressions is also defining a *semantics* for `Src`. We follow the same idea as in [2] and provide a denotational semantics for `Src`, written as a function `eval` which maps terms of `Src` to values.

```
⟦_⟧ : {t : Tyₛ} {z : Sizeₛ} → (e : Src t z) → Vec [[ t ]] z
⟦ vₛ v ⟧          = [ v ]
⟦ e₁ +ₛ e₂ ⟧      = [ ⟦ e₁ ⟧' + ⟦ e₂ ⟧' ]
⟦ ifₛ_thenₛ_elseₛ_ c e₁ e₂ ⟧ = if ⟦ c ⟧' then ⟦ e₁ ⟧ else ⟦ e₂ ⟧
⟦ e₁ ⟫ₛ e₂ ⟧      = ⟦ e₂ ⟧ +++ ⟦ e₁ ⟧

⟦_⟧' : {t : Tyₛ} {z' : Sizeₛ} → (e : Src t (suc z')) → [[ t ]]
⟦ e ⟧' = head ⟦ e ⟧
```

Figure 3: Semantics for the `Src` language.

Because of the sequencing construct, we chose our values to be *vectors* of naturals or booleans. The usage of vectors as values matches the requirement we imposed that sequences of expressions have to be uniformly-typed. Also, the size of the vector resulting from evaluation matches the size of the `Src` expression.

[1]Strict positivity implies that fields of a datatype can't be of function type with the respective datatype as its domain ('in negative position'). So constructors of the form $mkFoo : Foo \to \cdots \to Foo$ are not allowed.

A last interesting remark about `eval` is that the *type signature of eval expresses type preservation*. This phenomenon, of having proven a meta-theoretical property "for free", is allowed by the use of dependent types on the definitions of the language and the evaluator. On a non-dependent setting, such a property would have to be proven separately, in an "offline" proof.

Having defined `Src` and its semantics (`eval`), we now move on to defining the target language (`Bytecode`) and its corresponding semantics (`exec`). This target language is a *typed* stack bytecode, and therefore it operates on *typed stacks*:

```
StackType : Set
StackType = List Ty_s
```

Figure 4: The type of a `Stack`.

```
data Stack : StackType → Set where
   ε      : Stack []
   _▽_    : ∀ {t s'} → [[ t ]] → Stack s' → Stack (t :: s')
```

Figure 5: Definition of a *typed* stack.

The `Stack` type is *indexed* by `StackType`. The same `StackType` indices come into play in the definition of the `Bytecode` datatype: Each bytecode instruction performs a certain stack manipulation, and therefore each instruction is a value of type `Bytecode` $\mathtt{s}_1$ $\mathtt{s}_2$, where $\mathtt{s}_1$ is the *input StackType* and $\mathtt{s}_2$ is the *output StackType*.

```
data Bytecode : StackType → StackType → Set where
   SKIP : ∀ {s}          → Bytecode s s
   PUSH : ∀ {t s}        → (x : [[ t ]]) → Bytecode s (t :: s)
   ADD  : ∀ {s}          → Bytecode (ℕ_s :: ℕ_s :: s) (ℕ_s :: s)
   IF   : ∀ {s s'} → (t : Bytecode s s') → (e : Bytecode s s') → Bytecode (𝔹_s :: s) s'
   _⟫_  : ∀ {s_0 s_1 s_2} → (c_1 : Bytecode s_0 s_1) → (c_2 : Bytecode s_1 s_2) → Bytecode s_0 s_2
```

Figure 6: Typed bytecode instructions.

The semantics of the bytecode are defined in by the function `exec`, whose type signature can be read (due to *currying*) as mapping a value of `Bytecode` into a function which relates the initial pre-execution `Stack` to the final post-execution `Stack`.

```
exec : ∀ {s s'} → Bytecode s s' → Stack s → Stack s'
exec SKIP          s       = s
exec (PUSH v)      s       = v ▽ s
exec ADD           (n ▽ m ▽ s) = (n + m) ▽ s
exec (IF t e)      (true   ▽ s) = exec t s
exec (IF t e)      (false ▽ s) = exec e s
exec (c_1 ⟫ c_2)   s       = exec c_2 (exec c_1 s)
```

4 Compiler Correctness

With both source and target languages of our compiler in place, the compiler itself is now easily defined:

```
compile : ∀ {t z s} → Src t z → Bytecode s (replicate z t ++ₗ s)
compile (vₛ x)          = PUSH x
compile (e₁ +ₛ e₂)     = compile e₂ ⟫ compile e₁ ⟫ ADD
compile (ifₛ c thenₛ t elseₛ e) = compile c ⟫ IF (compile t) (compile e)
compile    ((ifₛ c thenₛ t elseₛ e) ⟫ₛ e₂) = ◦ (compile c ⟫ IF (compile t ⟫ compile e₂) (compile e ⟫ compile e
compile (e₁ ⟫ₛ e₂) = ◦ (compile e₁ ⟫ compile e₂)
```

Correctness of the compiler is defined as stating that `compile` should respect both evaluation and execution semantics: executing the code directly with the `eval` function should result in the same value as first compiling to `Bytecode` and then executing the `Bytecode` on an empty initial stack. Formally we need to proof the following:

```
correct : {t : Tyₛ} {z : Sizeₛ} {s' : StackType} (e : Src t z) (s : Stack s')
    → prepend ⟦ e ⟧ s ≡ exec (compile e) s
```

The proof follows by induction on the source language expression. The value case is proven using only beta reduction (and reflexivity). For the cases of addition and simple conditional (no suffix) we prove correctness by rewriting with the induction hypothesis and matching on the operands (in the case of addition) or the condition (in the conditional case).

The most complex case of the correctness proof is the sequencing case: here we use, besides the induction hypotheses over the operands, also a lemma involving prepending lists to vectors:

```
lemmaPrepend : ∀ {m n t st}
             → (v₁ : Vec [[ t ]] m) (v₂ : Vec [[ t ]] n) (l : Stack st)
             → prepend (v₁ +++ v₂) l ≅ prepend v₁ (prepend v₂ l)
```

Figure 7: Prepending a concatenation equals prepending the first list then the second.

Unfortunately, this lemma doesn't get us quite as far as we would want. What's left is a seemingly-not-so-difficult proof involving heterogeneous equality that doesn't lend itself to closure that easily. Although we're not really sure how to finish the proof, we've reduced it to two self contained holes that can be attacked independently.

5 Graph Representation

5.1 Introduction

The compiler in the version currently given generates bytecode where common substructures aren't explicitly shared. A piece of code sequenced after an if-then-else branching statement gets copied to both the true and the false branch. This gives the generated code a tree like structure which (allegedly) makes it easy to reason about. Unfortunately, the duplication results in exponential memory usage: for every branching statement appearing in the code, two copies of the code below it must be kept in memory. The following example shows this behaviour:

```
dupSource : Src ℕₛ 3
dupSource = if_s v_s true then_s v_s 2 else_s v_s 3 ⟫_s (v_s 5 ⟫_s v_s 7)
```

The following code is generated:

```
dupTarget : ∀ {s} → Bytecode s (ℕ_s :: ℕ_s :: ℕ_s :: s)
dupTarget = PUSH true ⟫ IF (PUSH 2 ⟫ PUSH 5 ⟫ PUSH 7) (PUSH 3 ⟫ PUSH 5 ⟫ PUSH 7)
```

One can see that the code the expression `v`$_s$ `5` $\rangle\rangle_s$ `v`$_s$ `7` is generated twice.

A solution to this problem is given in [1], where instead of as a tree, the bytecode is now represented as an acyclic graph. In this representation, structures can be given names which can be used to refer to the substructure. Only one copy of the structure is kept in memory, independent of the number of times this structure is referred to. This more efficient representation is nice if performance is a concern, but it arguably makes to harder to reason about. Ideally, one would like to write proofs using the tree represenation while making use of the graph representation when actually using the compiler. Luckily, [1] also describes how correctness proofs from the tree repesentation can be transferred ('lifted') to the graph representation. To make this lifting machinery work we first have to make the tree structure explicit.

5.2 Explicit Tree Structure

An explicit tree description of the `Bytecode` datatype is obtained by representing it as the fixed point of some functor. This makes the recursive positions explicit and will later allow us to label them. Because `Bytecode` is an *indexed* datatype, we represent it using an *indexed* functor. Fixed points of indexed functors are obtained by instantiating them using the following datatype:

```
record HTree {Ip Iq : Set} (F : (Ip → Iq → Set) → (Ip → Iq → Set) ) (ixp : Ip) (ixq : Iq) : Set where
  constructor HTreeIn
  field
    treeOut : F (HTree F) ixp ixq
```

One should notice that that this datatype doesn't actually meet the no-positivity requirement 2. We've locally disabled the no-positivity check as to allow us to continue work in this direction. We reasoned that instantiating `HTree` with a stricly positive functor would not allow any inconsistencies to leak out of the module. Unfortunately, disabling this check resulted in a number of compiler bugs (or maybe expected behaviour? we don't know) which resulted in stack overflows when querying types and memory exhaustion compiling certain functions. The compromises that had to be made to work around these problems will be addressed in the end of this section.

Using `HTree`, one can use the following functor

```
data BytecodeF (r : StackType → StackType → Set)
  : (StackType → StackType → Set) where
  SKIP' : ∀ {s}       → BytecodeF r s s
  PUSH' : ∀ {t s}     → (x : [[ t ]]) → BytecodeF r s (t :: s)
  ADD'  : ∀ {s}       → BytecodeF r (ℕ_s :: ℕ_s :: s) (ℕ_s :: s)
  IF'   : ∀ {s s′} → (t : r s s′) → (e : r s s′) → BytecodeF r (𝔹_s :: s) s′
```

to obtain an datatype isomorphic to the original: `∀ s s′ . Bytecode s s′ HTree BytecodeF s s′`

The isomorphism is witnessed by the following two functions:

```
treeIsoTo : {ixp ixq : StackType} → (code : Bytecode ixp ixq) → fromTree (toTree code) ≡ code
```

```
treeIsoFrom : {ixp ixq : StackType} → (tree : HTree BytecodeF ixp ixq) → toTree (fromTree tree) ≡ tree
```

The functions `compile` and `exec` have counterparts defined on this fixed point representation which are called `compileT` and `execT` respectively. Correctness for the tree representation can now be stated as follows:

```
correctT : ∀ {s t z} → (e : Src t z) → execT {s} (compileT e) ≡ prepend ⟦ e ⟧
```

One final word about trees: by defining trees we had to disable the positivity check. This resulted in bugs in agda that were triggered whenever a proof involved the fold function that accompanies `HTree`. To workout these issues, we had to omit the implementation of fold and postulate the relevant properties so that the typechecker would never actually have to inspect the `HTree` datatype. Proving things using these postulates is straightforward, albeit cumbersome.

5.3 Graph Structure

Our goal now is to define a graph structure which allows for automatic lifting of proofs from the tree structure. In the end we'd like a proof lifting function so that the correctness proof for graphs can be obtained as

```
correctG = lift correctT
```

Let us first start with the basics. A graph representation is obtained by the following modification of the `HTree` datatype:

```
data HGraph' {Ip Iq : Set} (F : (Ip → Iq → Set) → (Ip → Iq → Set) ) (v : Ip → Iq → Set) (ixp : Ip) (ixq : Iq)
  HGraphIn   : F (HGraph' F v) ixp ixq → HGraph' F v ixp ixq
  HGraphLet : (HGraph' F v ixp ixq) → (v ixp ixq → HGraph' F v ixp ixq) → HGraph' F v ixp ixq
  HGraphVar : v ixp ixq → HGraph' F v ixp ixq
```

In addition to constructors representing the recursive positions, there are two constructors for representing and using shared substructures. The `HGraphLet` constructor is used to give tag a given substructure, which can then be referred to by the `HGraphVar` constructor.

The observant reader will notice that the indices of the `HGraphLet` constructor aren't scoped correctly. This is a minor oversight which we only found out about last minute.

Bytecodes which use shared substructures can now be represented as values of the type `HGraph BytecodeF`. Every `HTree BytecodeF` can trivially be embedded in `HGraph BytecodeF` by ignoring the extra constructors. For graphs there are of course also the `execG` and `compileG` functions which define how to execute graph bytecode or compile to graph bytecode, respectively. Converting in the other direction however loses sharing information. Executing either the `unravel`ed graph or the original will still gave the same result.

Unfortunately, at this moment the previous example which was used to show code duplication cannot be rewritten in the more efficient graph representation because the graph representation has some of it's indices mixed up.

Correctness for graphs amounts to the following statement:

```
correctG : ∀ {s t z}
  → (e : Src t z) → execG {s} (compileG e) ≡ prepend ⟦ e ⟧
```

To prove this, we provide a framework which is generic over the base functor chosen, which in our case is `BytecodeF`.

5.4 Lifting Proofs

We've implemented a module called 'Lifting' which is parametrised by a correctness proof for trees and a proof that relates `compileG` to `compileT` via the `unravel` function. Concretely, to obtain a proof for `correctG`, one needs to open the module `Lifting` and supply it with `correctT` and following (only non-trivial) additional parameter:

$unravelLemma : \forall \{s\,t\,z\}\ unravelLemma : \forall \{s\,t\,z\}$
$unravelLemma : \forall \{s\,t\,z\}$
$\quad \to (src : \mathsf{Src}\ t\ z) \to compileT\ \{s\}\ src \equiv \mathsf{unravel}\ (compileG\ \{s\}\ src)$

The implementation of the `Lifting` module itself uses a generalization of the `Shortcut Fusion Theorem` which is mentioned but not proven in [1]. The generalization, although not in contradiction with our intuition, is not proven, and is thus one of the weak spots of our system.

To actually instantiate the `Lifting` module, one needs to supply a proof of the `unravelLemma` tailored for the specific functor at hand. Unfortunately, we didn't manage to supply a full proof of this lemma for our `BytecodeF` functor. There is only tiny hole left open, but unfortunately this hole asks us to construct an infinite type. The requested value must have type:

```
BytecodeF (HTree BytecodeF) ixp ixq → HTree BytecodeF ixp ix
```

The `HTreeIn` constructor has this type:

```
HTreeIn : ∀ {F} → F (HTree F) ixp ixq → HTree F ixp ix
```

But unfortunately we're not allowed to use it. The problem seems to be that normally such an infinite type cannot escape a datatype declaration: it is only allowed on the right hand side the declaration. Somehow, due to the our way of approaching the proof, this has leaked to the outside world. We think it is possible that a different approach would avoid exposing this type and thus allow for a full proof of this lemma.

6 Conclusions

In this project, we developed a provably correct expression compiler using dependent types in the Agda programming language. Also, we implemented a *systematic transformation* that, given the correctness proof for a "basic" compiler, can produce the correctness proof for a compiler based on the first but generating "optimized" (sharing-preserving) target code.

We believe that, even with such small languages as the ones we chose as examples, this methodology of systematically "extending" correctness proofs when adding compiler optimizations is an important area of research, and that using proof assistants based on dependent types is a promising approach.

To conclude the report, we would like to explain what are the points in which we do not consider our work to be finished. These are lemmas which we believe are true, but due to some technicality or sheer time limitations we were not able to prove completely:

1. The sequence clause ($\_\rangle\!\rangle\_$ constructor) of the basic (non-optimized) compiler is depending on a lemma that was left still to be proven.

    - This lemma (`lemmaPrepend`) involves vector concatenation and a "prepending" operation, but can only be stated (apparently) using heterogeneous equality, which hindered our attempts.

2. Whether the *Shortcut Fusion Law* for folds also holds over *indexed functors* is still a topic for further research. It is very plausible for this law to be true, but there is no proof, and any proof would most certainly be non-trivial. It is a very large gap in our framework which needs our first attention should this project be continued further.

3. The remaining part of the proof of the *Unravel Lemma* probably needs a different appoach. We don't think this will pose a serious problem.

4. `HGraph` still needs some generalization to allow for more general (read: non-useless) let bindings. At the moment they are not sufficient to allow us to implement the provided examples in their efficient graph form. This is a minor oversight which should require only a few (but technical) local changes to fix.

Finally, we would like to comment on some of the main difficulties we ran into during the development of this project. First of all, as already mentioned, we had to model in Agda concepts which do not fit easily the total setting, such as type-level fixpoint operators (violate the *strict positivity* requirement of Agda) and generic folds over functors (which do not pass the totality checker). To be able to use these definitions we had to enable compilation flags and pragmas which made programming in Agda less "secure". Enabling these flags exposed some agda bugs/expected behaviour which made our development experience less than pleasant. We had stack overflows when querying holes for their type and memory exhaustion when compiling programs. Pinpointing these problems and working around them took quite a bit of our time and we hope that future versions of agda will be more accommodating.

References

[1] Patrick Bahr. Proving correctness of compilers using structured graphs. FLOPS '14, to appear, February 2014.

[2] James Mckinna and Joel Wright. A type-correct, stack-safe, provably correct, expression compiler. In *in Epigram. Submitted to the Journal of Functional Programming*, 2006.