Compiling Exceptions Correctly in Agda

Daniël Heres
Rob Spoel

July 3, 2015

Contents

1 Introduction

Compilers transform computer programs from one representation, typically code in a programming language, to another, often some form of executable machine code. To run the program, we usually run the compiled code on a machine of the target architecture. However, it is also possible to evaluate the programming

language code directly. In a verified compiler, we prove that compiling and executing a program returns the same result as evaluating the code directly. We call this property *Compiler Correctness*.

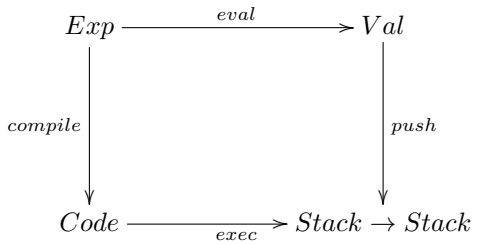
For our research project, we built such a verified compiler for a relatively simple programming language existing of typed expressions. Then we extended the language with exceptions, including mechanisms for throwing, catching and handling exceptions. We used the dependently typed programming language Agda for our implementation.

The goal was to find out how feasible different aspects of compiler verification are in a dependently typed setting. Where does the language help you, where does it get in the way?

2 Basic verified compiler

A widely cited paper on the topic of compiler correctness is *A type-correct, stack-safe, provably correct expression compiler in* EPIGRAM by James McKinna and Joel Wright [1]. In this article, McKinna and Wright describe the workings of a verified compiler for a very simplistic language. For their implementation, they used EPIGRAM, a language which is not widely used nowadays. Translating this work to Agda proved to be relatively straight forward.

The central idea of a correct compiler can be represented in a diagram:



The diagram must be interpreted as follows: Expressions can be compiled to machine code, which when executed results in some manipulation of the stack machine. Alternatively, we can recursively evaluate the expression to a value using denotational semantics, the evaluation function. If the expression can be totally reduced to a value, this value should be the same as the value that would have been placed on the stack by running the machine code.

This property can be expressed as such:

```
correct :: ∀ {s T} → (e : Exp T) → (st : Stack s)
  → eval e >> st ≡ exec (compile e) st
```

As can be seen, we give a parameter to the `Exp` constructor which indicates

it's type. Similarly, we enforce some properties for the `Stack` through the `s` parameter. These are signs that we are using the type system of Agda to aid us in proving this property. More about this later.

3 Our implementation with exceptions

In this section, we describe the details of how we implemented a verified compiler for a typed expression language with support for exceptions. We based our work on a paper by Graham Hutton and Joel Wright called *Compiling Exceptions Correctly*. [2] In this paper, Hutton and Wright explain the workings of implementing and verifying a a compiler that compiles exceptions and exception handlers through the clever usage of "tags" on the stack. Their implementation was in Haskell, which is not dependently typed. We took it upon ourselves to port these mechanisms to Agda.

In order to better understand our implementation, we thought it would be best to introduce aspects of it step by step.

3.1 Typed expressions

Our language consists of the following expressions:

```
data Exp : Type → Set where
  e-val : ∀ {T} → Val T → Exp T
  e-add : (e1 e2 : Exp NAT) → Exp NAT
  e-ifthenelse : ∀ {T} → Exp BOOL → (e1 e2 : Exp T) → Exp T
  e-throw : ∀ {T} → Exp T
  e-catch : ∀ {T} → Exp T → Exp T → Exp T
```

Most of these expressions should look familiar to most people. We can see the Agda type system in action right away. For those of us that are more Haskell-minded, this is not much different that a GADT. A final interesting remark is to note that the `e-throw` expression can be coerced to any type.

The denotational semantics of this language have an interesting property. Since expressions can not be guaranteed to evaluate to a value in the case of an uncaught exception, the return value of our evaluation function is `Maybe (Val T)`.

`e-val`, `e-add` and `e-ifthenelse` work as usual, returning their expected result as a `just x` as long as all the required sub-expressions (lazily) do not evaluate to `nothing`. As for the case of `e-throw`, it always evaluates to `nothing`. The `e-catch` expression tries to evaluate its first argument. If successful (not `nothing`), it returns it. Otherwise, it evaluates the second argument – the exception handler expression – and returns it.

It is immediately obvious that any case of a thrown exception is passed up the evaluation tree until it lands at a catching mechanism, which then runs the appropriate handler. This type of exception throwing and catching and handling might seem simplistic in the eyes of an experienced Java programmer, but should suffice as a proof of concept for the verification of the compiler of this language.

3.2 Virtual stack machine

3.2.1 Stack representation

Our virtual stack machine consists of two parts. One is the stack, and the other are the instructions that determine the stack manipulation.

The stack can contain three types of items. Any item on the stack is either a value (`val`), a handler tag (`hnd`) or a skip tag (`skp`). Encoded in the type of the stack is the type of the elements currently on the stack. This means that we can use dependently typed programming techniques to guarantee that certain instructions can only be run on stacks that contain the correct type of items at the top. Neat! For future reference, we call a cons-list of stack item types the `Shape` of the `Stack`. Our `Stack` data type has constructors to easily pattern-match on stacks of certain structures:

```
data Stack : Shape → Set where
  ϵ : Stack []
  _>>_ : ∀ {s T} → Val T → Stack s → Stack (val T :: s)
  hnd>>_ : ∀ {s} → {T : Type} → Stack s → Stack (hnd T :: s)
  skp>>_ : ∀ {s} → {T : Type} → Stack s → Stack (skp T :: s)
```

3.2.2 Code representation

```
data Instr : Shape → Shape → Set where
  PUSH : ∀ {T s} → Val T → Instr s (val T :: s)
  ADD : ∀ {s} → Instr (val NAT :: val NAT :: s) (val NAT :: s)
  COND : ∀ {s1 s2} → Code s1 s2 → Code s1 s2 → Instr (val BOOL :: s1)
      s2
  MARK : ∀ {s T} → Instr s (hnd T :: skp T :: s)
  HANDLE : ∀ {s T} → Instr (val T :: hnd T :: skp T :: s) (skp T :: s)
  UNMARK : ∀ {s T} → Instr (val T :: skp T :: s) (val T :: s)
  THROW : ∀ {s T} → Instr s (val T :: s)
```

These are the instructions that make up the stack machine language. `PUSH`, `ADD` and `COND` should look familiar to anyone who has dabbled into the realm of stack languages. The first pushes a new value onto the stack. The second takes the top two values from the stack and pushes the value gained by adding

them back onto the stack. The third reads the top item from the stack, and branches off to different code paths depending on its value. We can all see how our expressions would be compiled to these instructions.

The final four instructions are used to implement exception throwing and handling. The `THROW` instruction is obviously the machine code counterpart to the `e-throw` expression. A `e-catch` expression is compiled using the other three instructions.

Exception handling mechanisms are compiled by tagging the code as a piece of code that can handle exceptions. See below for a visualization on a bottom-growing stack:

`MARK`
Expression code here
...
`HANDLE`
Handler code here
...
`UNMARK`

When entering a block marked by `MARK`, the machine knows it can handle exceptions. In case an exception is thrown inside the expression code, the machine just needs to transition forward to the `HANDLE` instruction in order to start the exception handling routine. If the expression finishes without throwing however, the machine will need to skip the handler code all the way until the `UNMARK` tag. However you put it though, the `UNMARK` instruction indicates the end of this special area.

We use Agda's Star library to represent `Code` consisting of several chained instructions. Again, Agda's type system shines. By clever definition of our instructions, we have ensured that only valid combinations of instructions can be created by our compiler, since the instructions have the stack shape requirements encoded into their constructor types, and Agda's `Star` data type ensures the transitivity relation between these instructions.

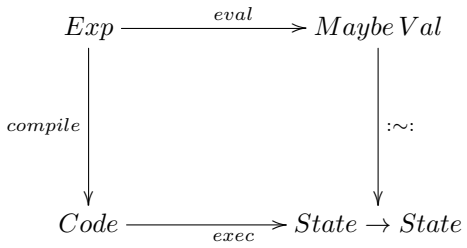
3.2.3 Executing code

From the exception catching mechanism, it is immediately clear we need three different execution states of our stack machine. In it's *normal execution state*, instructions are interpreted as normal. Whenever a `THROW` instruction is encountered, we move into the *exceptional execution state*. However, if no exception was thrown, the machine needs to be able to move past the handling routine by going into a *skipping execution state*.

In both of the special states, special care needs to be taken in the case of nested catch blocks. We do this by keeping a simple natural number counter

that we increase or decrease at the right time. For an exact implementation of the execution semantics, we refer you to our code.

Finally, it is a good moment to reflect back onto our original diagram from the second section. We have introduced the idea that expressions are evaluated to `Maybe Val`'s, and now we have introduced the idea that stacks are in fact stateful. This leads us to an updated version of the diagram. Compiler correctness is now represented as follows:



The :∼: operator is an operator that grows the stateful stack `State` by pushing a value onto the normal execution state in the case of `just x`, moving the state from the normal execution to the exceptional execution state in the case of `nothing`, and maintaining the state as-is in all other cases.

3.3 Proving Compiler Correctness

```
correct : ∀ {s T} (e : Exp T) (st : State s) → exec (compile e) st ≡ (
    eval e :∼: st)
```

This is the meat and potatoes of our work. By proving that this statement works for all expressions in all execution states, we would verify that our compiler outputs code that produces the same output as the denotational semantics of the expressions that were fed to said compiler.

Our implementation combines the languages of the two papers that we referred. This means that we had to redo the proof for the `e-ifthenelse` expression in this environment with a stateful stack execution machine. In order for the execution to be implementable, we were forced to execute one of the branches of the conditional even if we were in a special execution state. This was because Agda's type-checker was binding us to our strict types. This did not end up being a problem, since we were able to prove in a lemma that any executing any compiled expression in a special state of execution would not alter the state of execution in any way. See the `lemma-compiled-expr-maintains-_state` lemma's in our code.

Since the denotational semantic evalution of an `e-ifthenelse` expression did not require to evaluate one of the branches in the case of an exception being thrown – evident as the conditional subexpression evaluating to `nothing` – these

lemma's found some good use in proving the correctness of the compiler.

The other expressions were able to be proven through the use of equational reasoning. We refer the reader to our code for a more intrinsic look of the proofs. Agda as a proof assistant is clearer in conveying such mathematical verifications than the English language.

4 Conclusion

We showed the implementation of a verified compiler in Agda and we added exceptions to our language. In addition to that, we also showed that `e-ifthenelse` expressions are correct with respect to exceptions.

Implementing a basic language in a dependently typed language like Agda is relatively forward, but embedding a language feature like exceptions makes this task a lot more complex. A small oversight in the type definitions quickly propagates to the rest of the program. We found however that some of the more basic features in the Emacs mode for Agda such as the Auto command and equality reasoning made proving many of our statements much easier, especially considering it doesn't support tactics or an advanced automatic proving system at the moment of writing. That said, even though Agda was more than adequate for our task, some more features to make Agda more suitable for a proving assistant would be very welcome.

5 Future work

There are a lot of possible directions to extend our verified compiler. In this section we give some examples.

- Extending the language with more features, such as let bindings, functions and exception types.
- Using a more realistic language for a stack machine. For instance, it would be interesting to see someone prove compiler correctness in an implementation where the `COND` instruction can successfully be skipped during the special execution state. This might be achieved with a `JUMP` instruction.
- Also verify other parts of a compiler pipeline like optimizations, parsing and serialization that are part of a typical compiler.

6 Final remarks

During our implementation of this project, we hit many roadblocks. The final version of the code you see today has gone through several rewrites. Just

before the final rewrite, we were running into trouble because of our lack of understanding for the need of special execution states.

While looking for a solution to our problem, we ran into an existing implementation in Agda of the proofs presented in the paper by Hutton and Wright. We used this code to study the workings of the proofs. We were able to learn how to do proper equational reasoning in Agda thanks to this code.

Our final implementation strikes some resemblance to this existing code. We would like to take this moment to make clear that no fraud has taken place. We merely used the code to study and learn, and then implement our own version of the proofs based on it. Not a single line of code exists in our current version that we cannot explain and stand up for why it is necessary in this implementation.

Finally, our version includes conditionals, which were absent in the paper as well as the foreign implementation. This section merely exists as a full disclosure to come out in front of any possible misunderstandings.

References

[1] 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.

[2] Graham Hutton and Joel Wright. Compiling exceptions correctly. In Dexter Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 211–227. Springer Berlin Heidelberg, 2004.