Don't *bind* yourself to *do* notation!

Using Agda to Prove Correctness of Refactorings on Monads

Timen Zandbergen[1]

Supervisor(s): Jesper Cockx[1]**, Luka Miljak**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Timen Zandbergen
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Luka Miljak, Koen Langendoen

Abstract

This paper provides a refactoring from `do` notation to >>= operators and proves that this refactoring maintains observational equivalence. As programs grow ever larger and more complex, there is a growing need to automatically apply refactorings to these programs in a dependable manner. Current refactoring engines often contain errors, even if they are widely used and thoroughly tested. The methods used in the proof of contextual equivalence are described in this paper, with the goal of supporting future research into correct-by-construction refactorings.

1 Introduction

Refactoring is the practice of changing the code of a program without changing its observed behavior[1]. Refactorings are often employed to improve the quality of the codebase; it can make the code more testable, readable, extendable, maintainable, and so forth. It is vital that applying refactorings does not introduce new errors into the codebase.

For this reason, automated refactorings are a common feature of most Integrated Developer Environments (IDE) and Language Servers. IDEs are tool suites assembled around a text editor to provide language specific features such as error reporting and refactorings, and Language Servers decouple this feature set from any specific editor, allowing the features to be used by any editor that implements the Language Server Protocol[2]. These widely used and thoroughly tested refactorings are however not formally proven and contain an array of bugs [3]. The programmer can therefore not have absolute trust that applying an automated refactoring will preserve the soundness of their beloved code. By formally verifying these refactorings, we can guarantee no new errors will be introduced.

The correctness of software, and the extent to which this needs to be demonstrated, has long been a point of discussion within the field of computer science[4]. Errors in software have caused many severe accidents, including billions of dollars of economic damage and loss of life[5].

Both the 2021 and 2022 ACM Software System Awards have gone to teams working on formally verified systems[6]. They were respectively awarded to the teams behind CompCert, an optimizing compiler, and seL4, a high assurance microkernel operating system.

Many approaches exist to assure programmers of the correctness of their programs, each with various guarantees and trade-offs. Examples include regression testing and property based testing[7]. However, all these methods lack the full rigor of a formal proof. In addition, the seL4 project has shown that the cost of formal verification is lower than the cost of more traditional methods of engineering a high-assurance software system[8].

This paper provides a language specification of a Haskell-like language on which the refactoring is to be applied. Secondly the refactoring itself is described. Lastly a proof of equivalence pre- and post-refactoring and the steps to create it are described.

2 Background

In order to make this paper reasonably self-contained, this section will go over some of the tools and techniques used throughout the paper.

2.1 Agda

Both the refactoring and its corresponding proof of correctness have been written in Agda[9], a total, terminating, dependently typed language. This not only guarantees that our program won't loop forever nor throw errors, its dependent type system empowers us to write proofs checked during normal compilation.

A dependent type is a type that depends on the value of another type. One example would be `Fin n`, a type with n possible objects (0 through $n-1$), which can be used to structurally, at compile-time, guarantee that a number is smaller than n. This can be extended to types such as $x \equiv y$, which is a proof that the values of x and y are 'the same' according do the rules of the language. Additionally, $sym : x \equiv y \rightarrow y \equiv x$ is a proposition that, assuming x equals y, y equals x, with the implementation of this function being the proof of this proposition[10]. [1]

We can then use these 'propositions as types' to state properties about a program. For example, the proof that the program still evaluates to the same value after refactoring has a type similar to the following.

$$\forall \{M\ V\} \rightarrow (M \downarrow V) \rightarrow ((\text{refactor } M) \downarrow V)$$

Which states that any reduction trace from M to V can be converted into a reduction trace for the refactored M to that same value V, or more generally that refactoring the program doesn't change its resulting output. [2]

To be guaranteed that the implementation of a function is in fact a proof that the proposition of its type signature holds, the language must be *total*. This means that the language must be both terminating and cannot throw a runtime exception. If termination was not required, a function (read: proof) could simply call itself with the same arguments and never terminate but still inhabit that type. If the throwing of errors were possible, anything could be proven by simply throwing an exception, segfaulting, inducing a kernel panic, or other fun things.

2.2 Monads and their notation

A monad is a class of structures that wrap around values and encapsulate computational effects[11]. The notation `m A` means a monad that wraps around a type `A`. Monads implement two functions which specify their behavior. The `return` function wraps a single value in the monad of that type. The bind function (often noted as the infix operator >>=) takes in a monad wrapping type `A` and a function `A -> m B` and returns a monad wrapping type `B`. Their use cases will become more apparent throughout this section.

[1] In more classical terms, $x \equiv y$ *implies* $y \equiv x$.

[2] In actuality the proof is only of contextual equivalence as the function bodies in lambda expressions are also altered and these may be returned as values.

```haskell
halve :: Int -> Maybe Int
halve x = if even x
    then Just (x `div` 2)
    else Nothing

halveSum :: Int -> Int -> Maybe Int
halveSum x y =
  halve x >>= (\hx ->
  halve y >>= (\hy ->
  return (hx + hy)))
```

Figure 1: A Haskell program written using bind operators that sums the halves of two numbers, and returns `Nothing` if either number is odd.

```haskell
halveSum :: Int -> Int -> Maybe Int
halveSum x y = do
  hx <- halve x
  hy <- halve y
  return (hx + hy)
```

Figure 2: A Haskell program written using `do` notation that sums the halves of two numbers, and returns `Nothing` if either number is odd.

One prominent example is the Maybe type (often also called the Option type in other languages). A `Maybe T` is a type that either has or doesn't have an item of type `T`, it is often used to express the possibility of failure. It takes the place of null references or sentinel values and is less error-prone than them due to enforcing existence checks. Using the monad functions from before, we can specify a Maybe monad by having `return` be `Just` and having bind keep `Nothing` if the first input is `Nothing` else apply the function to the value inside the Maybe and return the result.

One advantage of using monads is that they can easily be chained together, avoiding the need for explicit control flow and bookkeeping. For example, a halving function that 'fails' if the input is odd can be used to create a function that halves both its inputs and returns their sum, but also fails if either of its inputs cannot be halved. The example in Figure 1 illustrates this combining with the bind operator.

But this style of composition can become clunky for larger chains, requiring quite a few lambda expressions and parenthesis. To solve this, Haskell has `do` notation, which provide a style of programming visually similar to imperative programming while remaining purely functional. Within a `do` block, the right-hand side of the `<-` is an expression resulting in a monad, and the left-hand side gets assigned elements within that monad one at a time. The result of the last expression is the result of the whole block and therefore also has to be a monad of the same type[12, p. 26]. This abstracts away the structure and allows the programmer to only consider one element at a time, as well as providing a more natural way to use monadic actions. Figure 2 is the same `halveSum` example, except now it's written using `do` notation.

In Haskell, this notation is desugared into a chain of bind operators, and the refactoring and proof presented in this paper focus on this equivalence.

A programmer might want to convert a chain of `do` blocks into their corresponding bind operators. One reason for this may include extracting the expression into a separate function allowing for code reuse.

3 The construction that is correct

This section describes the proof of correctness that has been produced, and what was created to reach that proof. First a description of the subset of Haskell and the modifications that were made is provided. Secondly the refactoring, and an implicit proof of well-typedness, is presented. Lastly the proof of semantic preservation is given.

3.1 The Haskell-like language

In order to demonstrate anything about the Haskell-like language (HLL) in Agda, its syntax and semantics must first be described in Agda.

In order to represent the language, an intrinsically well-typed representation was used. This means any program that type-checks as an Agda structure is also well-typed according to the typing rules of the HLL. This ensures that any constructed program is well-typed and obviates the need for separate well-typedness proofs.

De Bruijn notation was used to represent terms in the language[13]. This eliminates the need for string names and instead identifies terms using an index into the context. A number is used to represent how close the term is bound, with 0 representing the most recently bound term, and higher numbers representing values bound further away.

The actual syntax of this HLL was chosen to closely align with Haskell, although some deviations were made to accommodate easy embedding in Agda. This mainly involves swapping out symbols with preexisting meaning in Agda for visually similar symbols. The primary changes are that the backslash \ is replaced by a ƛ (lambda with stroke) to circumvent the special meaning of both the backslash character and the λ it visually represents. Additionally, the + and `*` are replaced by ✢ and ⋆ respectively to avoid collision with the number operators in Agda. Furthermore, a · rather than whitespace is used to specify function application. Lastly the `do` notation is expressed using `do<-` to avoid conflicting with the Agda `do` keyword and because the binding is not named anyway.

Two more limitations should be noted going forwards. Firstly only the `Maybe` monad is implemented due to the time constraints of this project. Secondly, rather than a `do` block with potentially multiple binding expressions, every `do<-` block creates a new block with the subexpression bound. These are equivalent due to the monad laws.

3.2 Refactoring do into bind

The actual refactoring that substitutes the `do` notation for bind operators (called `rmDo`) is given in its entirety in Figure 3 and due to intrinsic typing it also implicitly serves as a proof of well-typedness. For any language structure that is not `do` notation, recurse on every subexpression or if none exist simply return the expression. When `do` notation is encountered,

```
rmDo : Γ ⊢ A → Γ ⊢ A
rmDo (Term x) = Term x
rmDo (ƛ  L) = ƛ  (rmDo L)
rmDo (L · M) = (rmDo L) · (rmDo M)
rmDo (num x) = num x
rmDo (L ✢ M) = (rmDo L) ✢ (rmDo M)
rmDo (L ⋆ M) = (rmDo L) ⋆ (rmDo M)
rmDo true = true
rmDo false = false
rmDo (Nothing A) = (Nothing A)
rmDo (Just L) = Just (rmDo L)
rmDo (M >>= F) = (rmDo M) >>= (rmDo F)
rmDo (do<- M ⌒ F) = (rmDo M) >>= (ƛ
(rmDo F))
```

Figure 3: The refactoring that removes do notation from the program it is applied to.

✓: $\gamma \models$ L $\downarrow$ v $\rightarrow$ (rmDoEnv γ)$\models$(rmDo L)$\downarrow$(rmDoValue v)
With:

γ an environment over Γ

Γ a typing context

L a language construct ($\Gamma \vdash$ A)

A some type in the HLL

v some value of type A

Figure 4: The function signature of the proof of correctness. It proves that for any language construct that reduces to a certain value, the refactored version reduces to either the same value or a closure with the refactoring applied to it. It can be found in `refactoring.agda` as ✓.

a single instance of `do<- M ⌒ F` is replaced by `M >>= (ƛ F)`, with the refactoring also applied to the subexpressions `M` and `F`.

Two additional functions have been implemented, namely `rmDoValue` and `rmDoEnv`. The first one applies `rmDo` to the bodies of closures and `rmDoEnv` to the environments they enclose, and applies itself to the value inside a `Just` expression because that might also contain a closure, but leaves all other values untouched. The `rmDoEnv` function applies `rmDoValue` to all values in the environment, thus also only modifying closure values.

4 Proof of equivalence

Beyond well-typedness, the source contains a proof of equivalence called ✓, whose type signature can be found in Figure 4. It proves that, besides closure bodies, a reduction to a value can be mapped to a reduction from the refactored code to that same value. First this section explains why this signature was chosen. Secondly the techniques used to prove this statement are elucidated. Thirdly some considerations regarding defining reduction rules are given.

4.1 Observational equivalence

Given that the HLL has no side effects, it is referentially transparent, a proof of equivalence for the resulting value is sufficient to prove the equivalence of two programs. Unfortunately a proof of equality of the resulting values is not possible as the value may contain a closure wrapping a function body which has been refactored. We instead prove that these are observational equivalent, meaning that under any context, they will reduce to an equivalent value[14].

For this reason, `rmDoValue` is applied to the resulting value and `rmDoEnv` is applied to the environment. Both of these functions only affect closure values, and leave all others untouched. For closure values it simply applies the refactoring to their function bodies, the operation which we are currently proving preserves observational equivalence.

Furthermore, because HLL is terminating, (no fixpoint exists), and all closures must eventually return a non-closure value when enough arguments have been applied to it (it is not possible to construct a type of infinite length), and that non-closure value has been proven to be the same. We can therefore conclude that the closures returned by the refactored version are contextually equivalent.

Since the environment can contain closures, `rmDoEnv` needs to be applied to the incoming environment. These closures may or may not originate from a context which was not refactored, accordingly their function bodies may or may not have been refactored. Concretely, if in `L` a closure value `v` is taken from the environment, in `rmDo L` it might obtain `v` if the closure was created outside the refactored context, or `rmDoValue v` if it was also refactored. Because there is no way of knowing locally which terms have been refactored, refactoring all function bodies ensures that they have been.

4.2 Inhabiting the proof

This subsection will go over some of the methods used to prove the statement given in Figure 4.

Base case For all final reductions (↓`num`, ↓`lam`, and ↓`nothing`) except ↓`var`, the same reduction is returned. The language constructs they reduce are not affected by the refactoring and their reduction therefore remains the same. Mapping the reduction of a term (↓`var`), that is getting a value out of the environment, is discussed later this section.

Simple induction hypothesis For all reductions reducing constructs not modified by the refactoring, the same reduction is returned, but with ✓ recursively called on the subexpressions. This maps nicely to the recursive definition of `rmDo`.

Reducing do constructs In order to reduce what used to be `do<-` blocks, they are mapped to their equivalent bind reductions. For the case in which the `Maybe` contains `Nothing`, the reduction of the `Maybe` mapped using ✓ and the ↓`bindNothing` reduction is returned.

For the case in which the `Maybe` contains a value, the mapping requires slightly more work. The reductions from the `Mabye` to the `Just` value as well as the function body to value reductions can be mapped by calling ✓ recursively. The reduction trace from the second expression to the closure value however must be newly constructed. Since the refactoring wraps the second expression in a ƛ , this can be reduced using a single ↓`lam`. Agda automatically infers the function body of the reduction, along with the fact that the reduction of the resulting function can be created using the reduction of the original expression.

Taking values out of the environment The most subtle point of the proof was the mapping of ↓`var`, which represents taking values out of the environment. As mentioned earlier, the environment is modified by `rmDoEnv` because the value may or may not originate from outside the context which was refactored.

This conversion requires two intermediate steps. First, a proof that refactoring a value taken out of the original environment gives the same answer as taking out a value out of the refactored environment. This was constructed using standard structural recursion because the functions are defined in terms of each other. Second, a congruence proof is created that, given a proof of equality between two values and a reduction to the first value, returns a reduction to the second value. These two proofs are then combined to convert the reduction to one over the refactored domain.

4.3 Considerations for defining reduction rules

The HLL has its semantics described in terms of big-step reductions using environments. This section describes how these decisions made the proof of equivalence easier to construct. Firstly, the big-step reductions are compared to their small-step counterparts. Secondly, the choice of environments over substitution is expounded upon.

Reductions: big or small?

Under small-step reduction, a list of steps is given which all modify the program slightly until finally a value is reached. These reductions consist of two or three types of reductions[15]:

ξ**-reductions** reduce part of a term, possibly as specified by another reduction

β**-reductions** combine a constructor and a destructor; in this case lambda expressions and function application.

δ**-reductions** optional type of reduction that applies some built-in operation on values such as addition or multiplication

Under big-step reduction however, a tree of reductions is composed which more closely follows the syntax tree, and the reductions return values rather than simplified language constructs[15]. A new type of reduction is introduced, which converts primitive types to a value; such as reducing `num 5` to a value containing the number 5. Secondly, the ξ-reductions are incorporated into the β- and δ- reductions, which take reductions of their constituent terms and return a reduction of the whole to a value. For example, the ↓`add` reduction takes in the reduction to a number value of both the left and right sides, and returns the sum of these values.

Because the big-step reductions follow closely the recursive nature of how the refactoring is constructed, it becomes significantly easier to prove the equivalence pre- and post-refactoring. Additionally, because under big-step reductions the term has to reduce to a value rather than a simplified language construct, it becomes easier to state and prove equivalence of the resulting values.

```
data Ctx : Set where
  ∅   : Ctx
  _,_ : Ctx → Ty → Ctx

data _∋_ : Ctx → Ty → Set where
  Z : ∀ {Γ A}
    → Γ , A ∋ A
  S_ : ∀ {Γ A B}
    → Γ ∋ A
    → Γ , B ∋ A

data Env : Ctx → Set where
  ∅'   : Env ∅
  _,'_ : {Γ : Ctx} {A : Ty}
    → Env Γ
    → Value A
    → Env (Γ , A)

valueLookup : {Γ : Ctx} {A : Ty}
    → (γ : Env Γ)
    → (Γ ∋ A)
    → Value A
valueLookup ∅' ()
valueLookup (γ ,' x) Z = x
valueLookup (γ ,' x) (S y) = valueLookup γ y
```

Figure 5: The implementation of contexts, environments, and `valueLookup`, which given a proof that a value of type A exists in the context, returns that value. The implementation of `Ctx`, ∋, and `Env` are based on [15, Chapter Lambda and Chapter BigStep].

Environments over substitution

The second significant choice made was to use an environment to hold values rather than using substitution. This decision greatly simplifies the proof of equality, and its implementation is made easier by the well-typedness of the context.

The problem with trying to map reductions when using substitution is that the value substituted in might not be equal to the value substituted in under the refactored program. This means that not only does the locally considered reduction need to be mapped to the refactored program with recursive calls to their sub-reductions, a modification might also need to be made somewhere far deeper down the reduction tree to account for the different value being substituted in.

Using environments instead concentrates this difficulty into only the reduction that takes values out of the environment, a process already discussed in section 4.2.

Implementing the environments was a surprisingly clean process. The environments are indexed on a certain typing context, meaning that if the context associates a certain index to a type, the environment over that context contains a value of that same type. The value lookup function can therefore take an index into the context and use that to find a variable in the environment over that context. The implementation of `Ctx`, ∋, and `Env` are based on [15] chapters *Lambda* and *BigStep*, with `Env` having type information added. The `ValueLookup` was made to preserve type safety and to give an easy way to get values out of the environment given only the `Term` construct in the HLL. The exact definitions of these constructs is given in Figure 5.

5 Discussion

This section gives an overview of other methods of quality assurance, and discusses the limitations of formally proving correctness.

5.1 Other assurance methods

Traditional methods of testing programs, such as unit tests, integration tests, and property based testing, are often employed to assure a programmer of the correctness of their software. It is often noted, and has been since at least 1969, that testing a program only "shows the presence, not the absence of bugs"[16, p. 16].

A significant number of projects rely on their widespread adoption and a large amount of people reading over the code to assure users of their quality[17]. This tactic is often emulated within closed-source ecosystems by enforcing code review by a different programmer before merging code into the new branch[18].

However, even ubiquitous open-source programs have had catastrophic bugs in them. One of the most glaring examples of this was the Heartbleed vulnerability in OpenSSL[19], which despite operating between a quarter and half of popular HTTPS sites, carried an egregious security flaw for over two years before being discovered.

One study analyzing 198 failures in widely used software found that 77% of them could be reproduced by unit tests[20, Finding 9], with that number only being reached when the fault was already known to exist. Additionally, 38% of system specific catastrophic failures only occur in long-running systems, making the possibility space of input events intractable.

5.2 Limitations of formal proofs

It has been argued that formal proofs of correctness are an anachronistic way of solving the problem of errors in programs[21]. The basic idea is that the batch processing perspective is dominant in reasoning about software and programming languages, but that a significant portion of modern programs are long-lived 'live' programs such as operating systems or databases. For these types of programs, it is argued, the abilities to introspect and to update a running program are more important than proving some notion of correctness.

The Deep Space 1 spacecraft is one example of formal verification failing[22]. The program had been proven to be free from concurrency errors, but this proof relied on the assumption that only the concurrency constructs in the specially designed language would be used. The person assigned to write a part of the code was not properly informed of this and called into some lower level Lisp constructs, thus unknowingly invalidating the proof. The probe deadlocked twenty hours after becoming autonomous, but was revived by dynamically updating the Lisp code running on it.

5.3 Limitations of the proof

This subsection will discuss some of the limitations of the proof that is presented in this paper. Additionally, possible remedies will be given where it is considered feasible.

Firstly, it is important to note that the refactoring and the resulting proofs were done on an intrinsically well-typed syntax tree rather than actual source code. This means that the input program must be well typed, and no behavior is specified for ill-typed programs. Furthermore, the conversion to and from text is done using Agda's parser and printer, a complete proof could also prove some correspondence between the text representation and the underlying syntax tree.

Secondly, the proof and refactoring are specified in relation to a small subset of a 'real' programming language that uses semantics that are similar but not equivalent to real Haskell. Given that the refactoring only transforms the program where `do` notation is used, extending the language with additional orthogonal constructs should only result in trivial changes needing to be made. However, a distinction in semantics does exist as `do` notation in Haskell also includes pattern matching and `fail` recovery, which are not present in my language due to the time constraints of this research project.

The specification of the HLL semantics in terms of Agda is not very problematic, however, due to the similarity of the languages[23], and it is common to use deep embedding to represent a language in a proof assistant[24].

6 Responsible Research

The source code (and therefore proofs) used in this paper is available online[3], and can be verified as correct by loading it in Agda. No attempt has been made to subvert the correctness guarantees of Agda. All code has been written in a manner which should make its function evident, and as such can be safely built upon.

The limitations of my approach have been documented in the Discussion section. This is done such that the reader can make an informed judgement on whether to reproduce and expand the ideas presented in this paper, or to embark on other methods of quality assurance.

A lot of work has been done on proving formal correctness by actors not everyone would agree are ethical. One example would be seL4, which, although made in Australia, was funded by DARPA through Boeing, to create an "optionally manned helicopter". The morality of the United States Military Industrial Complex is left as an exercise to the reader. Conversely, there is also a lot of interest in high assurance software in fields such as medicine. And software errors in such fields have been disastrous for many people.

7 Conclusions and Future Work

This paper has presented a refactoring that converts `do` blocks into chains of >>= and has proven the result to be observationally equivalent. The way this proof was constructed, the subset of Haskell that has been used to write it upon, and

[3]https://github.com/MetaBorgCube/brp-agda-refactoring-timenzandberge

the choices made in the creation of this Haskell-like language have been explained.

Supporting more of Haskell Currently, the subset of Haskell that is included in the HLL is very limited. Expanding this refactoring to all of modern Haskell would allow it to be used in real-world projects. Broadening the proof to include orthogonal constructs can be done analogously to current constructs not affected by the refactoring. There would be a challenge in describing the semantics for all possible monads, as these can also be user defined.

Supporting more refactorings By combining this correct-by-construction refactoring with other refactorings that have also been proven correct, a powerful suite of refactorings can be composed. These smaller refactorings could then also be combined into larger refactorings that implicitly carries all the properties of the smaller refactorings they are composed of[25].

A correct parser and printer Currently, the proof of correctness is only valid for an implicitly well-typed abstract syntax tree. Future work would need to be done to combine this language description with a parser and pretty-printer which are also proven correct. The proof of correctness of the refactoring could then be extended to include the actual string representations pre- and post-refactoring.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999, ISBN: 0201485672.

[2] H. Bünder and H. Kuchen, "Towards multi-editor support for domain-specific languages utilizing the language server protocol," in *Model-Driven Engineering and Software Development*, S. Hammoudi, L. F. Pires, and B. Selić, Eds., Cham: Springer International Publishing, 2020, pp. 225–245, ISBN: 978-3-030-37873-8.

[3] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, "Systematic testing of refactoring engines on real software projects," in *ECOOP 2013 – Object-Oriented Programming*, G. Castagna, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 629–653, ISBN: 978-3-642-39038-8.

[4] E. W. Dijkstra, "On the cruelty of really teaching computer science," *Communications of the ACM*, Dec. 1988. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF.

[5] J. Snijders, C. Morrow, and R. van Mook, *Software Defects Considered Harmful*, RFC 9225, Apr. 2022. DOI: 10.17487/RFC9225. [Online]. Available: https://www.rfc-editor.org/info/rfc9225.

[6] L. Zhou, V. Issarny, C. Fournier, T. Schlossnagle, and C. Thekkath, *ACM Software System Award*, 2022. [Online]. Available: https://awards.acm.org/software-system.

[7] M. Aniche, *Effective Software Testing: A developer's guide*. Simon and Schuster, 2022, ISBN: 9781633439931.

[8] G. Klein, J. Andronick, K. Elphinstone, *et al.*, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, Feb. 2014, ISSN: 0734-2071. DOI: 10.1145/2560537.

[9] Agda Development Team, *Agda 2.6.3 documentation*, 2023. [Online]. Available: https://agda.readthedocs.io/en/v2.6.3/.

[10] P. Wadler, "Propositions as types," *Commun. ACM*, vol. 58, no. 12, pp. 75–84, Nov. 2015, ISSN: 0001-0782. DOI: 10.1145/2699407.

[11] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '92, Albuquerque, New Mexico, USA: Association for Computing Machinery, 1992, pp. 1–14, ISBN: 0897914538. DOI: 10.1145/143165.143169.

[12] S. Peyton Jones *et al.*, "The Haskell 98 language and libraries: The revised report," *Journal of Functional Programming*, vol. 13, no. 1, Jan. 2003, http://www.haskell.org/definition/.

[13] N. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem," *Indagationes Mathematicae*, vol. 75, no. 5, pp. 381–392, 1972. DOI: 10.1016/1385-7258(72)90034-0.

[14] A. Ahmed and M. Blume, "Typed closure conversion preserves observational equivalence," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '08, Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 157–168, ISBN: 9781595939197. DOI: 10.1145/1411204.1411227.

[15] P. Wadler, W. Kokke, and J. G. Siek, *Programming Language Foundations in Agda*. Aug. 2022. [Online]. Available: https://plfa.inf.ed.ac.uk/22.08/.

[16] B. Randell and J. Buxton, "Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee," Oct. 1969. [Online]. Available: https://eprints.ncl.ac.uk/file_store/production/55593/5C8B829F-C598-48F6-8726-0EF473BB7338.pdf.

[17] E. Raymond, "The cathedral and the bazaar," *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[18] D. J. Reifer, "How good are agile methods?" *IEEE software*, vol. 19, no. 4, pp. 16–18, 2002.

[19] Z. Durumeric, F. Li, J. Kasten, *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14, Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488, ISBN: 9781450332132. DOI: 10.1145/2663716.2663755.

[20] D. Yuan, Y. Luo, X. Zhuang, *et al.*, "Simple testing can prevent most critical failures: An analysis of production failures in distributed Data-Intensive systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO:

USENIX Association, Oct. 2014, pp. 249–265, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan.

[21] J. Rusher, "Stop Writing Dead Programs," presented at Strange Loop 2022, Saint Louis, Missouri, Oct. 2022. [Online]. Available: https://youtu.be/8Ab3ArE8W3s.

[22] M. L. Klaus Havelund and S. Parka, "Formal analysis of the remote agent before and after flight," in *Lfm2000: Fifth NASA Langley Formal Methods Workshop*, National Aeronautics and Space Administration, Langley Research Center, vol. 210100, 2000, p. 163. [Online]. Available: https://web.archive.org/web/20111019054900/http://ti.arc.nasa.gov/m/pub-archive/176h/0176%20(Havelund).pdf.

[23] J. Cockx, O. Melkonian, L. Escot, J. Chapman, and U. Norell, "Reasonable agda is correct haskell: Writing verified haskell using agda2hs," in *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, ser. Haskell 2022, Ljubljana, Slovenia: Association for Computing Machinery, 2022, pp. 108–122, ISBN: 9781450394383. DOI: 10.1145/3546189.3549920.

[24] A. Šinkarovs and J. Cockx, "Extracting the power of dependent types," in *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2021, Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 83–95, ISBN: 9781450391122. DOI: 10.1145/3486609.3487201.

[25] D. Horpácsi, J. Kőszegi, and S. Thompson, "Towards trustworthy refactoring in erlang," in Proceedings of the Fourth International Workshop on *Verification and Program Transformation,* Eindhoven, The Netherlands, 2nd April 2016, G. Hamilton, A. Lisitsa, and A. P. Nemytykh, Eds., ser. Electronic Proceedings in Theoretical Computer Science, vol. 216, Open Publishing Association, 2016, pp. 83–103. DOI: 10.4204/EPTCS.216.5.