Toward Type-Preserving Compilation of Coq

William J. Bowman
Northeastern University, USA
wjb@williamjbowman.com

Abstract

Type systems have been adopted by many languages as a means of lightweight verification of language invariants. Languages with *dependent types* like Coq allow programmers to encode rich program invariants that are automatically checked by the type system, supporting program verification. However, even a verified program in Coq may result in errors if it is compiled incorrectly or linked against target code that violates the program invariants assumed in the source program.

Our work develops techniques to verify compilers for languages like Coq and specify when linking a compiled program will not result in errors. We develop a *closure conversion* translation, a common compiler pass for functional languages, that preserves types into the compiler intermediate languages (ILs). We compile a significant fragment of the core language of Coq into a dependently typed IL. Having types in the IL supports compiler verification and gives a simple way of specifying what target code a compiled program may be safely linked with. We prove that this translation satisfies *compositional compiler correctness*, which states that the linking a program with Coq libraries and running the result in Coq is the same as separately compiling the program and its required libraries, then linking, and running the result in the IL.

ACM Reference format:
William J. Bowman. 2017. Toward Type-Preserving Compilation of Coq. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Problem and Motivation

Type systems are the most widely used lightweight program verification system. Standard type systems can enforce *language invariants* such as "integer division is never performed on strings".

Languages with *dependent types*, like Coq, support expressing rich *program invariants* in the type system. Types in these languages can be viewed as specifications. If a program is well typed then the program is a proof that its specification is correct. This expressive power comes from the ability for types to *depend* on programs. A type can refer to program variables and can have code embedded in it. For instance, consider the following type for a function f which takes two arguments, x and y, both of which are natural numbers.

$$f : (x : Nat, y : Nat) \to (x + y = y + x)$$

The result type of f expresses that $x + y$ is equal to $y + x$. Notice that the type of f refers to the *values* that f takes as arguments. The type also contains the programs $x + y$ and $y + x$. These are programs using the standard addition function and not some separate logical version addition used only in types. This type can be viewed as a specification stating that addition is commutative and the function f is a proof of this specification.

Dependently typed languages like Coq have been used to verify safety-critical programs. Appel (2015) use Coq to verify the OpenSSL implementation of SHA-256, a widely used cryptography primitive. The CompCert optimizing C compiler is written and verified in Coq (Leroy 2009).

Unfortunately, even programs written and proven correct in a dependently typed language like Coq can exhibit certain kinds of errors when running: *miscompilation errors* and *linking errors*.

A *miscompilation error* occurs when an error in the compiler introduces errors into the program being compiled. For instance, suppose there is an error in common subexpression elimination, a standard compiler optimization.

```
if (...) {...; x = y + 1; ...}
else     {...; x = y + 1; ...}
```

The compiler may try to optimize the above code by moving `x = y + 1` out of each branch, reducing code size. However, if the optimization fails to check that `x` or `y` are not modified in any of the elided snippets of code (the `...`s) then this optimization may change the behavior of the program, resulting in an error. In the

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

case of Coq, programs are often compiled to OCaml—a strongly typed functional language with imperative features. OCaml itself is compiled to machine code. This introduces two opportunities for miscompilation errors in a Coq program.

A *linking error* occurs when a source-language *component*—a program that needs to be linked with other code before it can be run—is compiled and linked with target-language components that violate some source-level program invariant. For instance, if a verified Coq component is compiled to OCaml and linked with OCaml code that use imperative features, the resulting program may crash. This is because the verification engine in Coq implicitly assumes that functions behave *purely functionally*, *i.e.*, that they always return the same value when given the same inputs. This is not true in OCaml where functions can modify memory and return values depending on both the inputs to the function and the values stored in memory. When an imperative OCaml function is linked with code compiled from Coq, the imperative function can easily violate program invariants from Coq. For instance, if the Coq program assumes that a function will always return a particular data structure on a particular input then the Coq-to-OCaml compiler may generate optimized code that is specialized to this data structure. However, if the imperative function ultimately linked with the Coq program returns a number indicating an error code, then the verified Coq program may end up dereferencing an arbitrary memory location.

Our work seeks to rule out miscompilation errors and linking errors for programs written in dependently typed programming languages like Coq. We do this by studying *type-preserving compilation* for dependently typed languages. A *type-preserving compiler* preserves type information from the source language into the intermediate languages (ILs) used by the compiler. The types can support compiler verification and can be used to specify when it is safe to link compiled components. If we could preserve dependent types through compilation then we could generate components from Coq and safely link with OCaml components as long as each OCaml function has a type that guarantees that it behaves purely functionally.

2 Background and Related Work

Type-preserving compilation has been widely studied in the context of traditional—*i.e.*, non-dependent—type systems. Early work preserves types only through early phases of compilation, enough to support optimizations for polymorphic/generic code (Leroy 1992; Tarditi et al. 1996). Later work preserves types to a Typed Assembly Language (TAL) (Morrisett et al. 1998). TAL guarantees that linking any well-typed components results in a type- and memory-safe program. Safety is guaranteed even when linking with unknown TAL code. Recent work uses type-preserving compilation to support compiler verification. Ahmed and Blume (2008) use type-preserving compilation to prove that a closure conversion translation preserves all equivalences from the source language into the target language. This kind of result guarantees that certain security and data hiding properties are preserved by the compiler. Bowman and Ahmed (2015) translate a language that guarantees information flow security into a language with a standard type system, using types to prevent linking with attackers that violate the information flow policies of the source language. Perconti and Ahmed (2014) use types to define safe linking between a high-level source and low-level target language—akin to using inline assembly in C—and verify a compiler from the high-level source to the low-level target while allowing linking with any target code that is safe to link with.

Certifying compilation has been studied by using limited forms of dependent types in compiler ILs. A certifying compiler is not verified, but it produces a proof of correctness for each binary. This proof can be independently checked to show that the binary was compiled correctly, removing the compiler from the trusted code base. Early work compiles a type- and memory-safe subset of C to assembly and produces proofs that the assembly code is also type- and memory-safe (Necula and Lee 1998). This compiler encodes proofs using LF, a simple dependently typed language that can only express first-order formulas. Later work uses *indexed types* to support the same proofs but with a smaller trusted code base (Xi and Harper 2001). Indexed types support encoding only decidable formulas from a particular theory—in this case, a subset of integer arithmetic sufficient to reason about memory accesses. Recent work uses type-preserving compilation to enforce end-to-end security properties from Fine—a language with *refinement types*, a limited form of dependent types that allow expressing types as decidable subsets of values—to a refinement-typed version of the .NET intermediate language (Chen et al. 2010).

In contrast to these restricted forms of dependent types, Coq supports so-called *full-spectrum dependent types*. *Full-spectrum dependent types* make the language of types and programs the same—a type can be written in and computed by the *full* programming language. This allows expressing near-arbitrary predicates over programs. The predicates themselves can be undecidable, but generally the type system ensures that checking a proof is still decidable. Recall that in a dependently typed language, programs are proofs and types are specifications, so this is the same as saying that type checking is decidable.

The only work studying type-preserving compilation of full-spectrum dependent types has shown negative results or restricted the language to be unusable in practice. Barthe et al. (1999) show that the continuation-passing style (CPS) translation, a common compiler pass for functional languages that makes control flow explicit, is type-preserving when restricted to a variant of Coq in which even type checking is undecidable. Barthe and Uustalu (2002) show that in the presence of *strong dependent pairs*, a common and useful data type expressed by full-spectrum dependent types, the CPS translation is not type-preserving. Further investigation finds that adding unrestricted `goto`-like control operators to a language with strong dependent pairs causes the proof language to become logically inconsistent (Herbelin 2005).

3 Uniqueness of Approach

Our approach is unique in that we neither give up decidable type checking nor limit the expressivity of the types. The major challenge to accomplishing this is translating specification and proofs in a way that maintains validity and decidable checking. As we translate programs, we are making implicit high-level language invariants explicit in the low-level code. We must also express those invariants in the specifications and automatically adapt the proofs.

In this work we study closure conversion. Closure conversion is a common compiler pass used when implementing functional programming languages. In a functional language, a function can refer to any variables in the current scope. As we compile to machine code, functions need to be translated into blocks and allocated in memory. In order to lift a function with free variables into the global scope and then allocate it in memory, we first package all the free variables from the function's current scope into a data structure called the *environment* and then package the environment with a *closed* version of the function which has no free variables. Below we give an example of such a translation.

$$\begin{aligned}\mathsf{fun}\,(x).\,e \rightsquigarrow \{&\mathsf{env} = \{x_1 = x_1, \ldots, x_n = x_n\};\\ &\mathsf{m} = \mathsf{fun}\,(env, x).\\ &\qquad x_1 = env.x_1;\\ &\qquad \ldots;\\ &\qquad x_n = env.x_n;\\ &\qquad e\}\end{aligned}$$

The notation $e \rightsquigarrow e'$ means that the closure conversion translation of e is e'. We write $\mathsf{fun}\,(x).\,e$ to mean an anonymous function of one argument, x, whose body is e. We translate the function into a structure with two fields, an environment field and a method field. This data structure is called a *closure*. For the environment, we create a structure containing all the free variable that appear in e. For the method, we project all variables from the environment before running the body of the original function. Any time the function is called, the environment in `env` is passed in as an additional argument. Now the function is easily translated into a memory-allocated block. The closure data structure will end up containing a dynamically allocated structure for the environment and a statically known function pointer. This translation essentially turns a functional language into a simple object-oriented language. Each closure can be thought of as an object with one method and some arbitrary fields (the environment).

The translation just described is defined on an untyped language. In our setting, we must also translate types for functions into types for closures. In the function type $f : (x : A) \to B$, both A and B can refer to any free variables in scope when the specification is written, *i.e.*, any variables in the environment. The problem is that the closure's environment is meant to be private while the closure's specification is meant to be public. We want a client to read the specification, but we don't want to expose private information about the environment.

Past work on type-preserving closure-conversion uses *existential types* to quantify over the type of the environment thus keeping its structure hidden from the client (Minamide et al. 1996). Each specification would be translated from $f : (x : A) \to B$ to $f : \exists P.(env : P, x : A) \to B$. This new specification says that for this closure there exists an environment of type P such that when the closure is invoked with that environment and an argument of type A, then the result has type B.

There is a problem when we try to use this translation in a dependently typed language. In the source language, A and B can refer to free variables in the current scope. However, after this translation they cannot refer to free variables now stored in the environment since the type of the environment is entirely hidden by the existential quantification. Minamide et al. (1996) notice a similar problem in the presence of polymorphic (*i.e.*, generic) types, where the types can contain type variables. They use *translucent types* to add additional equalities about the environment into the specification. Translucent types express a locally scoped equality in the type system. This allows expressing equalities about how the environment is related to free variables. Later work observed that since type variables could be treated differently than program variables, a far simpler translation suffices to handle polymorphism (Morrisett et al. 1998). Unfortunately, these simplifications do not apply to dependently typed languages since types can now contain program variables, not just type variables.

We adapt the approach of Minamide et al. (1996) to dependent types. We translate $f : (x : A) \to B$ to

$f : \exists P, \exists v.(env = v : P, x : A) \to B$. In addition to existentially quantifying over the type of the environment, we also quantify over a *value* that the environment is equal to; this equality is the translucent type. The subtle part of this translation is that we don't change A or B to refer to env. Instead, we leave them with references to free variables. The proof for this specification will have to provide a specific v. The rules for checking a translucent type allow the type system to unify the free variables from A and B with the v from the proof.

To demonstrate this, let us consider the following example. We closure convert a function that takes a natural number and returns an even natural number. The type of the function refers to the free variable is_even

$$
\begin{array}{l}
f : (y : \mathsf{Nat}) \to \mathsf{is\_even}(y) \rightsquigarrow \\
\quad \mathsf{exists}(\{P = \mathsf{Struct}; \\
\qquad\qquad v = \{is\_even = \mathsf{is\_even}; \dots\}; \\
\qquad\qquad m = \mathsf{fun}(env : \mathsf{Struct}, y : \mathsf{Nat}) \dots\}) \\
\quad \mathsf{as}\ \exists P, v.(env = v : P, y : \mathsf{Nat}) \to \mathsf{is\_even}(y)
\end{array}
$$

When we translate f, we produce a closure containing the environment and the translated function. We also declare that it has an existential type $\exists P, v.(env = v : P, y : \mathsf{Nat}) \to \mathsf{is\_even}(y)$ and provide witnesses for P and v. We have only exposed is_even in the type, even though the environment may contain other free variables. It can be confusing that the environment simply creates a structure where the field is_even is assigned the value of the variable is_even; this seems to do no useful work. However, recall that the function assigned to m will be lifted into a global scope, while the environment and the declared type of the closure remain in the current scope. Therefore this assignment stores the current value of is_even in memory and passes that value to the function in the environment field is_even.

To check that the translated program is well typed, we first infer a type for the function assigned to m. The type of the function on its own is $(env : \mathsf{Struct}, y : \mathsf{Nat}) \to env.is\_even(y)$. The type system must now prove that this type is equal to the declared type. When checking the values under an exists, the type system replaces the existential variables with their witnesses. We need to show that the type we inferred is compatible with the declared type $(env = \{is\_even = \mathsf{is\_even}; \dots\} : \mathsf{Struct}, y : \mathsf{Nat}) \to \mathsf{is\_even}(y)$. Translucent types give a principle for specializing a function argument to a specific value, therefore we can show that $env : \mathsf{Struct}$ and $env = \{is\_even = \mathsf{is\_even}; \dots\} : \mathsf{Struct}$ are compatible. Now it suffices to show that $env.is\_even$ is the same as is_even. This is simple since the *value* of env is present in the type.

4 Results and Contributions

We develop the first verified and type-preserving closure conversion translation for a full-spectrum dependently typed language. We compile the Calculus of Inductive Constructions (CIC), a significant fragment of Coq, to a dependently typed IL. We prove compositional correctness of the translation and show that types are preserved into the IL. The typed IL provides a specification for safe linking in the IL. Linking well-typed components is guaranteed to result in a type-safe program. The particular translation is general and applies to a large class of dependently typed languages, and the proof techniques apply to other type-preserving translations. Furthermore, the closure conversion translation provides insights about how to preserve high-level concepts from a dependently typed language into the low-level concepts of a machine language.

The essence of our translation is given in Figure 1. We translate each function into a closure, resulting in all functions in the IL being closed and ready to lift into the global scope. We hide the type of the environment using existential types and unify the type of the closed function with the declared type of the closure using translucent types. We translate each function type into an existential type describing the closure.

Key to showing that this translation is type preserving is showing that we preserve *definitional equivalence*. The type system relies heavily on a decidable equivalence between programs, written $e \equiv e'$, called *definitional equivalence*. This allows the type system to run programs in the types and to solve equalities such as those used in the translucent type. We show that our closure-conversion translation preserves this equivalence.

Lemma 4.1 (Preservation of Equivalence). *If* $e_1 \equiv e_2$ *and* $e_1 \rightsquigarrow e_1'$ *and* $e_2 \rightsquigarrow e_2'$ *then* $e_1' \equiv e_2'$.

Recall that since Coq has full-spectrum dependent types, programs and types are the same. This lemma shows that we preserve equivalence between programs and equivalence between types. To show that the translations of function types are still equivalent, we essentially rely on the argument we used in the example at the end of §3. To show that we translate equivalent functions into equivalent closures, we develop a novel principle of *normal form* for closures and an algorithm for computing the normal form. A *normal form* is a syntactic form of the program that is syntactically identical to all other programs that should be considered equivalent. This reduces equivalence of programs to checking that they are syntactically identical.

The proof that closure conversion is type preserving is fairly involved due to the complexity of Coq's type system, but the final statement of type preservation is simple.

$$
\begin{array}{l}
\mathsf{fun}(x : A).e \rightsquigarrow \mathsf{exists}\{P = \mathit{Struct}(x_1 : A_1, \ldots, x_n : A_n); \\
\qquad\qquad\qquad\qquad v = \{x_1 = x_1, \ldots, x_n = x_n\}; \\
\qquad\qquad\qquad\qquad m = \mathsf{fun}(\mathit{env} : \mathit{Struct}(x_1 : A_1, \ldots, x_n : A_n), x : A'). \\
\qquad\qquad\qquad\qquad\qquad\qquad x_1 = \mathit{env}.x_1; \\
\qquad\qquad\qquad\qquad\qquad\qquad \ldots; \\
\qquad\qquad\qquad\qquad\qquad\qquad x_n = \mathit{env}.x_n; \\
\qquad\qquad\qquad\qquad\qquad\qquad e'\} \\
\qquad\qquad\qquad\quad \mathsf{as}\ \exists P, v.(\mathit{env} = v : P, x : A') \to B' \\
\text{where } A \rightsquigarrow A' \\
\qquad e \rightsquigarrow e' \\
\qquad x_1 : A_1, \ldots, x_n : A_n = \mathsf{free\text{-}vars}(e') \\
\\
(x : A) \to B \rightsquigarrow \exists P, v.(\mathit{env} = v : P, x : A') \to B' \\
\text{where } A \rightsquigarrow A' \\
\qquad B \rightsquigarrow B'
\end{array}
$$

Figure 1. Closure Conversion Translation for CIC (excerpts)

Theorem 4.2 (Type Preservation)**.** *If e is a well-typed program and $e \rightsquigarrow e'$, then e' is also well-typed.*

In addition to showing that the translation is type preserving, we show *compositional compiler correctness*. We define an interpreter for Coq programs and another for IL programs. We show that the result of linking components in Coq and running the result using the Coq interpreter is equivalent to compiling the components, then linking them, and running the result using the IL interpreter. This rules out *miscompilation errors* for this pass of the compiler.

Theorem 4.3 (Compositional Compiler Correctness)**.** *If e is a well-typed component that expects to be linked with components of types $A_1, \ldots, A_n$, and we have components of these types, $e_1 : A_1, \ldots, e_n : A_n$, and $e \rightsquigarrow e'$ and $e_i \rightsquigarrow e_i'$ then the result v of interpreting e linked with $e_1, \ldots, e_n$ is equivalent to the result v' of interpreting e' linked with $e_1', \ldots, e_n'$.*

Note that we defined linking in terms of the types of the components. This guarantees that, even though we know nothing about the components beyond their types, the result of linking is always type safe. Since our languages are dependently typed, a component's type can even express sophisticated pre-conditions on what it can be linked with. This rules out *linking errors* for this stage of the compiler.

Our results here are only an initial step toward a verified type-preserving compiler for Coq. The next step in this compiler is to memory allocate functions and global data structures. Then we need to design a dependently typed assembly language. The memory allocation translation should be straightforward, although we must be careful not to introduce logical inconsistency. Recall that Coq disallows any kind of imperative features. Once we add explicit memory operations to our ILs, we must also add invariants that prevent using memory from introducing logical inconsistency. This will become harder to solve in a dependently typed assembly language, where the only features are essentially imperative. We should be able to adapt prior work on Typed Assembly Language, which gives a type system that allows restricting memory operations to behave more like values in a functional (Morrisett et al. 1998). The harder problem is that we must find a way to restrict jumps to prevent logical inconsistency. Prior work shows that features similar to unrestricted `goto` cause logical inconsistency, but also demonstrates one method for restricting these features to regain consistency (Herbelin 2005). In recent work, Patterson et al. (2017) have shown how to design a typed assembly language where assembly components must return to their callers, which effectively rules out unrestricted jumps (goto). This may serve as a starting point for a dependently typed assembly language.

References

Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *ICFP 2008*. 12. https://dl.acm.org/citation.cfm?id=1411227

Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2, Article 7 (April 2015), 31 pages. DOI:https://doi.org/10.1145/2701415

Gilles Barthe, John Hatcliff, and Morten Heine B. Sørensen. 1999. CPS translations and applications: the cube and beyond. *Higher-Order and Symbolic Computation* 12, 2 (1999), 125–170. DOI: https://doi.org/10.1023/A:1010000206149

Gilles Barthe and Tarmo Uustalu. 2002. CPS translating inductive and coinductive types. In *PEPM 2002*. Association for Computing Machinery (ACM). DOI:https://doi.org/10.1145/503032.503043

William J. Bowman and Amal Ahmed. 2015. Noninterference for Free. In *Proceedings of the 20th ACM SIGPLAN International*

Conference on Functional Programming (ICFP 2015). ACM, New York, NY, USA, 101–113. DOI:https://doi.org/10.1145/2784731.2784733

Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving Compilation of End-to-end Verification of Security Enforcement. In *PLDI 2010*. 412–423. DOI:https://doi.org/10.1145/1806596.1806643

Hugo Herbelin. 2005. On the degeneracy of Σ-types in presence of computational classical logic. In *International Conference on Typed Lambda Calculi and Applications*. 209–220. DOI:https://doi.org/10.1007/11417170_16

Xavier Leroy. 1992. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 1992*. 177–188. DOI:https://doi.org/10.1145/143165.143205 arXiv:http://gallium.inria.fr/ xleroy/publi/unboxed-polymorphism.pdf

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *POPL 1996*. http://dx.doi.org/10.1145/237721.237791

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. From System F to Typed Assembly Language. In *POPL 1998*. http://dx.doi.org/10.1145/268946.268954

George C. Necula and Peter Lee. 1998. The design and implementation of a certifying compiler. In *PLDI 1998*, Vol. 33. ACM, 333–344. DOI:https://doi.org/10.1145/277652.277752

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: Reasonably Mixing a Functional Language with Assembly. In *PLDI, Barcelona, Spain*.

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-Language Semantics. In *ESOP 2014*. http://dx.doi.org/10.1007/978-3-642-54833-8_8

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. 1996. TIL: A Type-Directed Optimizing Compiler for ML. In *PLDI 1996*. 181–192. DOI:https://doi.org/10.1145/231379.231414

Hongwei Xi and Robert Harper. 2001. A Dependently Typed Assembly Language. In *ICFP 2001*. 169–180. DOI:https://doi.org/10.1145/507635.507657