Editable shallow embeddings: A pragmatic approach to software system verification

ANTAL SPECTOR-ZABUSKY, University of Pennsylvania, USA
JOACHIM BREITNER, DFINITY Foundation, Germany
YAO LI, University of Pennsylvania, USA
STEPHANIE WEIRICH, University of Pennsylvania, USA

If a code base is so big and complicated that complete formal verification is intractable, can we still apply and benefit from formal verification methods? We show that by allowing a deliberate *mechanized formalization gap* we can shrink and simplify the model until it is manageable, while still retaining a meaningful, declaratively documented connection to the original, unmodified source code. Concretely, we translate core parts of the Haskell compiler GHC into Coq, using `hs-to-coq`, and verify invariants related to the use of term variables.

Additional Key Words and Phrases: Haskell, Coq, Compiler Verification

1 INTRODUCTION

Consider a large, mature software system, such as the Glasgow Haskell Compiler (GHC). Is there any hope of applying formal verification to it? Clearly, complete verification is out of the question, as GHC is both too *big* and too *complicated* in all its many tightly intertwined gritty details. But maybe we can get some of the benefits of formal verification – such as bug discovery, checked and explicit documentation of invariants, and a deeper understanding of the code base – if, instead of attempting to *completely* verify all of the details of GHC, we deliberately carve out a specific portion to focus our efforts on.

In order to even begin, we need a version of GHC's source code that allows for mechanized reasoning. The `hs-to-coq` tool [Spector-Zabusky et al. 2018] can be used to provide a shallow embedding of Haskell code into Gallina, the functional programming language of the Coq proof assistant [Coq development team 2004], where we can reason about it using Coq's logic. For example, Breitner et al. [2018] use this embedding to show that the `Data.Set` and `Data.IntSet` modules from the `containers` library are correct implementations of finite set data structures.

Will this work for GHC? To answer this question, we have developed a case study in compiler verification. Our goal is to translate a part of GHC related to the intermediate language Core into Gallina, formalize invariants related to the representation of term variables in that language, and prove that selected transformations preserve these invariants.

This case study differs from prior uses of `hs-to-coq` in that we are working with code drawn from a complex system, not a library. Because we focus on *part* of this system, our embedding cannot start at the roots of the dependency hierarchy. Verifying GHC from these roots down until finally we reach the portion of interest would take too long; we need a way to identify and reason about code in the middle of the dependency graph. Furthermore, because we are working with a complex system, we need a way to simplify some but not all of the details, focusing our attention on the aspects of the system that are relevant for our case study.

Authors' addresses: Antal Spector-Zabusky, Computer and Information Science, University of Pennsylvania, 3330 Walnut St, Philadelphia, PA, 19104, USA, antals@cis.upenn.edu; Joachim Breitner, DFINITY Foundation, Germany, joachim@dfinity.org; Yao Li, Computer and Information Science, University of Pennsylvania, 3330 Walnut St, Philadelphia, PA, 19104, USA, liyao@cis.upenn.edu; Stephanie Weirich, Computer and Information Science, University of Pennsylvania, 3330 Walnut St, Philadelphia, PA, 19104, USA, sweirich@cis.upenn.edu.

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

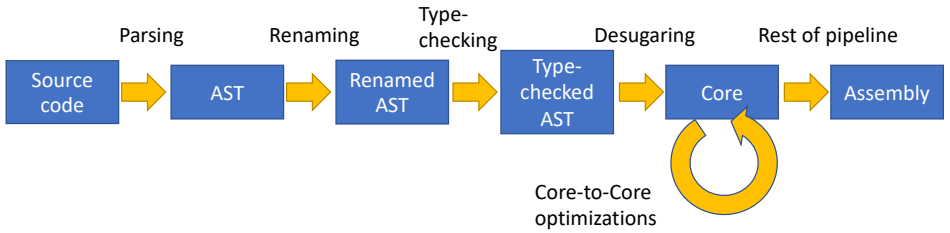

Fig. 1. GHC compilation pipeline

Conveniently, the design of `hs-to-coq` relies on *edit files*, instructions that guide and control the translation from Haskell into Gallina. Previous work only used these edits to make the code acceptable to Coq, and sought to replicate in Gallina the semantics of the original Haskell code as much as possible. In this work, we have increased the expressiveness of the edit language to permit more liberal translations.

When translating GHC, we use edits to elide or abstract whole modules, data types, and functions that aren't related to the parts of the code that we would like to reason about. These edits select *parts* of the full system, even though full verification is impractical. Furthermore, in the code that we do want reason about, we use edits to hide information, simplify the development, and reduce the possibilities that we need to consider. These edits *simplify* parts of the system, rather than trying to preserve its full generality.

Of course, we do not want to abstract and simplify everything. A compiler implementation like GHC relies on subtle properties of its data structures, and it is exactly these invariants that we would like to study with the help of a proof assistant. We want the power of selective attention, so that we can focus on certain details and be blissfully ignorant of many others.

The advantage of this approach over defining a Gallina model of the system by hand is that we get a *mechanized formalization gap*. Our model of GHC is mechanically connected to the original source by a repeatable process, and all simplifying assumptions are recorded in the edits files and can be inspected there.

In this paper, we report on the success of our case study, which demonstrates that this approach is tractable and pragmatic. In particular, our work makes the following contributions:

- We extend `hs-to-coq` with new forms of edits that allow for extensive simplification of the translated Gallina code (Section 3).
- We present a Gallina model of a subset of GHC, derived from the original Haskell source code, and abstract and refine its representation of Core language expressions and variables to ensure that it is suitable for verifying properties of term variable binding (Section 4).
- We develop a formal specification of two Core invariants, well-scopedness and the correct use of join points, that should be preserved by operations and optimizations (Section 5).
- We prove properties about free variable calculation and substitution related to these invariants and show that the exitify optimization pass preserves both invariants (Section 6).

All of our work is available online under the open-source MIT license, including the Core language model, the edits required for its creation, and the proofs of its properties.[1]

2 CASE STUDY: THE GLASGOW HASKELL COMPILER

The case study that we have selected for this project is a portion of the Glasgow Haskell Compiler (GHC).[2] Specifically, GHC performs most of its optimization work on an intermediate language called *Core*; we selected as our target the goal of reasoning about the usage of variables during these Core-to-Core passes. This intermediate language is the target of GHC's type inference and desugarer, as shown in Figure 1.

Why does GHC make a good case study? First, GHC is one of the largest open source Haskell projects available. The version that we target in this project (GHC 8.4.3) contains over 180 000 lines of code.[3] It is a realistic example of a complex system and a suitable challenge for our methodology. At the same time, this challenge is self-contained. Because GHC is a bootstrapping compiler, it relies only on a small number of external libraries.

Second, GHC is a mature project. The first prototype was built thirty years ago [Hudak et al. 2007]. The current design of the Core language dates from the mid-2000s [Sulzmann et al. 2007], but is still the target of significant revisions [Breitner et al. 2016; Maurer et al. 2017]. Although the compiler has been around a long time, it is under active development by a large, distributed team of contributors. (One co-author of this paper is a GHC developer.) The code itself is well documented, both internally (there are over 100 000 lines of comments) and externally (there is a wiki documenting the compiler,[4] and the source repository contains a guide to the design of Core [Eisenberg 2015]).

This internal documentation is our source for the scoping and join point invariants that are the target of our case study (see Section 2.2). These invariants come directly from comments in the source code in GHC itself and from the implementation of the `CoreLint` invariant checker. These invariants are significant to the GHC developers, and because of the relative simplicity of the Core language, they are easy to specify (see Section 5). Yet, at the same time, they cause difficulties for property-based testing [Pałka et al. 2011]. The code that must maintain these invariants is subtle and designed for performance [Peyton Jones and Marlow 2002]. Yet, even though some parts of this implementation have been in GHC for over twenty years, we know of no attempt to mechanically verify of its correctness.

2.1 The Core AST in Haskell and Gallina

The Haskell version of the Core AST is shown on the left side of Figure 2.[5] This language is based on an explicitly-typed variant of System F called System FC [Eisenberg 2015; Sulzmann et al. 2007]. It includes variables (`Var`), constant literals (`Lit`), function applications (`App`), lambda abstractions (`Lam`), potentially-recursive let bindings (`Let`), and case expressions (`Case`). The `Tick` constructor marks profiling information and the last three data constructors carry information related to the type system. Note that the Gallina version renames the `Var`, `Type` and `Coercion` data constructors, as Coq uses only a single namespace for types and values. Additionally, as `Type` is a keyword in Coq, `hs-to-coq` automatically renames the Haskell type named `Type` by adding an underscore.

GHC goes to significant effort to maintain a *typed* intermediate language, and not just to allow optimizations to be guided by types. When run in developer mode, GHC checks that the output of each Core-to-Core pass is well-typed. Core is explicitly typed, so this requires only a single pass,

[1] https://github.com/antalsz/hs-to-coq
[2] Available from https://www.haskell.org/ghc/
[3] All statistics about lines of code given in this paper refer to non-blank, non-comment lines, and have been calculated using the `cloc` tool, available from https://github.com/AlDanial/cloc.
[4] https://gitlab.haskell.org/ghc/ghc/wikis/commentary/compiler/
[5] This and all other code samples in the paper may be reformatted, have module names or comments removed, or similar for greater clarity.

```
data Expr b
   = Var Id
   | Lit Literal
   | App (Expr b) (Arg b)
   | Lam b (Expr b)
   | Let (Bind b) (Expr b)
   | Case (Expr b) b Type [Alt b]
   | Cast (Expr b) Coercion
   | Tick (Tickish Id) (Expr b)
   | Type Type
   | Coercion Coercion
  deriving Data
data Bind b
   = NonRec b (Expr b)
   | Rec [(b, (Expr b))]
  deriving Data
type Arg b = Expr b
type Alt b = (AltCon, [b], Expr b)
type Id = Var
type CoreBndr = Var
type CoreExpr = Expr CoreBndr
type CoreBind = Bind CoreBndr
type CoreProgram = [CoreBind]
```

```
Inductive Expr b : Type
 := | Mk_Var : Id -> Expr b
    | Lit : Literal -> Expr b
    | App : (Expr b) -> (Expr b) -> Expr b
    | Lam : b -> (Expr b) -> Expr b
    | Let : (Bind b) -> (Expr b) -> Expr b
    | Case : (Expr b) ->  b -> Type_
      -> list ((fun b_ => (AltCon * list b_ * Expr b_)) b)
      -> Expr b
    | Cast : (Expr b) -> Coercion -> Expr b
    | Mk_Type : Type_ -> Expr b
    | Mk_Coercion : Coercion -> Expr b
with Bind b : Type
 := | NonRec : b -> (Expr b) -> Bind b
    | Rec : list (b * (Expr b)) -> Bind b.

Definition Arg := Expr.
Definition Alt := fun b_ => (AltCon * list b_ * Expr b_).
Definition Id := Var.
Definition CoreBndr := Var.
Definition CoreExpr := (Expr CoreBndr).
Definition CoreBind := (Bind CoreBndr).
Definition CoreProgram := (list CoreBind).
```

Fig. 2. Haskell (left) and Gallina (right) versions of the Core AST.

called `CoreLint`, which also checks a number of additional syntactic Core invariants. The GHC developers report that this design was crucial to find many tricky bugs [Marlow and Peyton Jones 2012]. Under normal compiler operation, `CoreLint` is not executed, but it can always be enabled with a command-line option. For us, `CoreLint` is a source for precise specifications of Core invariants.

The `Expr` data type is parameterized by the type of bound variables, as can be seen in the `Lam` case. For the part of the code that we consider in this paper, this type is `Var`; other parts of the compiler, which we do not interact with in this work, use a different type for variable bindings. The `Expr` data type is already rather succinct; the desugarer converts the much larger source Haskell AST[6] into Core by elaborating the many forms of syntactic sugar found in Haskell. One source of brevity is the reuse of the `Lam` constructor for abstraction over terms, types, and coercions. Similarly, GHC uses the `Var` type to represent term, type, and coercion variables. When used as term variables, as in the `Var` case of `Expr`, variables are called identifiers and are referred to by the type synonym `Id`.

2.2 Core operations and invariants

Our case study focuses on the properties of syntactic operations (calculating free variables and substitution) and the *exitify* optimization (which transforms code so that the compiler has more opportunities to inline definitions).

We selected these operations for our case study because of familiarity (one co-author is the developer of `exitify`), because they do interesting things to terms without relying on types, and because there are two interesting Core invariants that they non-trivially preserve:

[6]The `HsExpr` data type has 40 data constructors.

- *Well-scopedness*: All local variables are in the scope of a matching binder. This fundamental property is a prerequisite for terms to even make sense.
- The *join point invariants*, which describe where and how *join points* (local jump targets) may be declared and called. This is interesting because join points are innovative and a relatively new addition to GHC [Maurer et al. 2017].

Since our case study only involves reasoning about operations that manipulate terms, we also wish to limit our reasoning to the usage of term variables and avoid reasoning about types and coercions altogether. This focus is pragmatic; although the term language of Core is delightfully simple, the type and coercion languages are not. Having to reason about these data structures would significantly increase our work. This focus is also justified; the operations that manipulate types and coercions are often independent of the term language operations, are implemented in separate modules in GHC, and maintain their own set of invariants.

Having such focus is essential. The operations that we would like to consider are just a small part of GHC. In fact, the `CoreFVs`, `CoreSubst` and `Exitify` modules that contain the relevant implementations are each under 500 lines of code. In contrast, the entire source code of GHC 8.4.3 includes 464 Haskell modules and 182 174 lines of code.

However, this focus is also challenging. As mentioned above, GHC uses the same data structure `Var` to represent term, type and coercion variables. Furthermore, the free variable and substitution operations work simultaneously over all three sorts of variables. Our simple solution to this issue is to reduce the generality of our model. We use edits that modify the representation of variables so that they just cannot represent type and coercion variables (see Section 4), thus ensuring that we never need to reason about them in our work.

3 THE `HS-TO-COQ` EDIT LANGUAGE

In order to translate the portions of GHC we needed to prove our theorems about Core, we have added new functionality to `hs-to-coq`. These features have been inspired by our evolving approach to translation (e.g., our extended use of axiomatization and code modification in support of proof development) as well as by the unique features of the GHC code base.

In this section, we describe these features in more detail. However, to provide context, we first summarize the existing capabilities of `hs-to-coq` edits and their typical use in translation.

Building on the existing capabilities of `hs-to-coq` edits. Prior work on translating the `containers` library demonstrated the capabilities of `hs-to-coq` for medium-scale programming [Breitner et al. 2018]. At this scale, it was already necessary for users of the tool to define a number of edits in service of mechanical verification. These edits included:

- Removing Haskell features that make no sense in a shallow embedding, including `reallyUnsafePtrEquality#` and `seq`, using **`rewrite`** edits.
- Skipping parts of the code that are irrelevant for verification or are difficult to translate (such as code related to serialization and deserialization), using **`skip`** edits.
- Modifying the representation of integers to avoid reasoning about overflow, using **`rename type`** edits.
- Substituting operations that are difficult to reason about (e.g., bit twiddling functions) with simpler definitions, using **`redefine`** edits.
- Managing recursive functions that were not structurally recursive by either providing hints for the termination proof along with the definition or by deferring those proofs altogether, using **`termination`** edits.

The translation of GHC also requires all of these previously-extant edit forms. However, we found that this functionality was not enough. Therefore, we have added the following new edit

forms to `hs-to-coq`. In the next section, we describe in more detail how we make use of these features when simplifying them to the Core expression data type and its operations.

Constructor skipping. The Core AST, though simpler than source Haskell, carries around a great deal of information that is not germane to our verification goals. Some of this information is in the form of metadata; some of it is in the form of type and coercion variables that we do not analyze. Regardless, since our model of GHC does not concern itself with these properties, we would like to avoid dealing with these concerns. However, the problematic cases are often *sub*cases of other data types – for example, the `Expr` data type for Core contains a case `Tick` strictly for profiling information, as we see in Figure 2. Thus, we added support for a new **`skip constructor`** edit that eliminates an entire case from data types.

For example, the edit **`skip constructor`** `Core.Tick` removes the `Tick` constructor from of `Expr` and then propagates this information to delete any equation of a function definition or arm of a case statement that matches, directly or indirectly, against this constructor. More dramatically, we use this edit to modify the representation of variables, which we discuss further in Section 4.1.

In this way, the development of the embedding gives us assurance that our code of interest is independent of particular features of GHC, without doing any proofs. In particular, if the targeted code needed to use the skipped constructor in a fundamental way (e.g., if it were used to construct a value in some operation that could be skipped), then the output of `hs-to-coq` would not be accepted by Coq. We can only skip constructors that we can isolate.

We do need to be careful, however – heavy use of **`skip constructor`** can lead to wildcard cases that no longer match anything, as all the would-be "extra" constructors have been skipped. Because Coq does not permit redundant pattern matches, we also add an edit to manually delete such cases. The edit **`skip equation`** `f pat1 pat2 ...` removes the equation matching `pat1 pat2 ...` from the definition of the function `f`.

Axiomatization. The ability to axiomatize definitions and modules was added to `hs-to-coq` directly for this project. There are many definitions in GHC that we don't want to translate for one reason or another. However, sometimes this code is used within the other functions that we want to verify, but not on a code path that is exercised by our proof (for example, in a metadata update). We thus provide an **`axiomatize definition`** edit, which replaces any value in Haskell with a Coq **`Axiom`** at the same type. In contrast to a **`redefine`** edit, axiomatization is more limited – we make fewer assumptions about the behavior of the edited operation. This edit was extremely valuable during the translation of GHC, as it also allowed for incremental verification.

We offer multiple ways to interact with axiomatization. For example, while we are focused on Core, its dependencies transitively reach into huge portions of GHC, and we don't want to deal with all of them. While some modules we can skip (via **`skip module`**), this isn't always viable. For example, the `FastStringEnv` module declares a type for maps keyed by GHC's `FastString` type. These are used, for example, when manipulating metadata for data type constructors, but this is not an operation we need to concern ourselves with to verify properties of variables. We can thus **`axiomatize module`** `FastStringEnv`, which leaves the type definitions intact and automatically axiomatizes every definition in the module as per **`axiomatize definition`**.

We also want to make use of axioms to replace *type* definitions. As type definitions do not have kind annotations, we cannot automatically generate axioms; we instead use the **`redefine`** or **`rename`** edits to replace one definition with another. For example, **`redefine`** `Axiom DynFlags.DynFlags : Type` replaces a record of configuration options with an opaque axiom.

While being able to axiomatize Haskell definitions is important, it does have the potential to introduce inconsistency – if we axiomatized the Haskell definition `undefined :: a`, we would be

able to prove any theorem we wanted. As a result we need to examine the functions we axiomatize; however, in GHC, most functions are not fully polymorphic and return inhabited types.

We discuss the use of axiomatization further in Section 4.5, with a focus on its specific importance for extracting a slice of GHC. However, even though we automatically axiomatize many definitions in our development, we almost never manually add axioms about their properties. We discuss why this is the case in Section 7.1.

Mutually recursive modules. GHC, nearly uniquely among Haskell programs, makes significant use of *recursive* modules; most of the modules that define Core are part of a single mutually-recursive cycle. This is not a feature supported by Coq, so as part of the translation, we have had to introduce edits that combine multiple source modules (both translated and axiomatized) into a single target. We use this facility to create the module `Core`, which contains the definition of the abstract syntax of the Core intermediate language. (For more, see Section 4.5.)

Mutual recursion edits. The Core AST that we saw in Figure 2 is defined via the mutually inductive types `Expr` and `Bind`. Typically, operations that work with either of these data structures are perforce mutually recursive. For example, consider the `exprSize` operation, shown in the left hand side of Figure 3; it is mutually recursive with the analogous function `bindSize`. These functions compute the size of an expression and a binder, respectively.

Coq can natively show the termination of mutually defined fixed points, as long as they are each recursive on one of the mutually recursive types and make recursive calls to each other on strict subterms of their arguments. One important pattern this naive treatment of termination prohibits is "preprocessing" – recursion that goes through an extra function which simply forwards to one of the recursive functions. This is something we see in as simple a definition as that of `exprSize` and `bindSize`. Along the way, the function `altSize` is called, which simply unpacks a tuple and recurses into `exprSize` and `bindSize`. From Coq's perspective, that means *all three* of these functions are mutually recursive, and one of them has as its only argument a tuple. And there's a fourth function, `pairSize`, which has the same problem.

Since a tuple isn't a mutually recursive inductive data type, for Coq to accept the definitions of `exprSize` et al., we must *inline* the definitions of `pairSize` and `altSize` into the mutually recursive definitions that use them. To tell `hs-to-coq` to do this, we use the **`inline mutual`** edit:

```
inline mutual CoreStats.pairSize
inline mutual CoreStats.altSize
```

This results in the Coq definition on the right in Figure 3, as well as new free-standing *non*-recursive definitions of `pairSize` and `altSize` which are the same as their (local) `let`-bound definitions.

Partiality. Prior work has either avoided partial operations altogether [Spector-Zabusky et al. 2018], or attempted to isolate them behind total interfaces [Breitner et al. 2018]. That isn't possible with GHC – many more operations may fail, for a number of reasons. At the same time, we only really care about *total* compiler code. If a compiler fails to produce an output for some input, then a compiler correctness proof says nothing.

One source of partiality comes from the use of GHC's operation for signaling a run-time error (i.e., a compiler bug) – the function `Panic.panic`. We cannot translate this function, since it actually throws an exception (using `unsafeDupablePerformIO`, no less). Instead, we axiomatize this operation as follows:

```
Axiom panic : forall {a} `{GHC.Err.Default a}, GHC.Base.String -> a.
```

The `Default` class, introduced by Breitner et al. [2018], is a class for types with at least one inhabitant, which is used with opacity to avoid any dependence on what the specific inhabitant

```
exprSize :: CoreExpr -> Int
-- ^ A measure of the size of the expressions,
-- strictly greater than 0
exprSize (Var _)         = 1
exprSize (Lit _)         = 1
exprSize (App f a)       = exprSize f + exprSize a
exprSize (Lam b e)       = bndrSize b + exprSize e
exprSize (Let b e)       = bindSize b + exprSize e
exprSize (Case e b _ as) =
  exprSize e + bndrSize b + 1 + sum (map altSize as)
exprSize (Cast e _)      = 1 + exprSize e
exprSize (Tick n e)      = tickSize n + exprSize e
exprSize (Type _)        = 1
exprSize (Coercion _)    = 1

bindSize :: CoreBind -> Int
bindSize (NonRec b e) = bndrSize b + exprSize e
bindSize (Rec prs)    = sum (map pairSize prs)

pairSize :: (Var, CoreExpr) -> Int
pairSize (b,e) = bndrSize b + exprSize e

altSize :: CoreAlt -> Int
altSize (_,bs,e) = bndrsSize bs + exprSize e
```

```
Definition exprSize : CoreExpr -> nat :=
  fix exprSize (arg_0__ : CoreExpr) : nat :=
    let altSize (arg_0__ : CoreAlt) : nat :=
          let 'pair (pair _ bs) e := arg_0__ in
            bndrsSize bs + exprSize e
  in match arg_0__ with
     | Mk_Var _ => 1
     | Lit _ => 1
     | App f a => exprSize f + exprSize a
     | Lam b e => bndrSize b + exprSize e
     | Let b e => bindSize b + exprSize e
     | Case e b _ as_ =>
       ((exprSize e + bndrSize b) + 1) + sum (map altSize as_)
     | Cast e _ => 1 + exprSize e
     | Mk_Type _ => 1
     | Mk_Coercion _ => 1
     end
with bindSize (arg_0__ : CoreBind) : nat :=
   let pairSize (arg_0__ : (Var * CoreExpr)%type) : nat :=
         let 'pair b e := arg_0__ in
           bndrSize b + exprSize e
   in match arg_0__ with
      | NonRec b e => bndrSize b + exprSize e
      | Rec prs => sum (map pairSize prs)
      end
for exprSize.
```

Fig. 3. Mutual recursion in Haskell (left) and Gallina (right)

is. Instances of it are mostly autogenerated by `hs-to-coq`. This constraint therefore enforces that `panic` can only be called with a return type that is known to be inhabited, ensuring that it does not introduce unsoundness. Although `panic` does not terminate the entire Coq program as it does in Haskell, arriving at it in a proof terminates our ability to reason about the code. Therefore, proving properties about code that uses `panic` also increases our confidence that it will not be triggered on that code path.

Partiality turns out to be important for translating GHC, much more so than we realized going in. For example, Haskell can define record selectors for single constructors of data types with multiple branches; these record selectors are thus necessarily partial.

While the `Default` class is not new, we have made much more significant use of it here due in particular to GHC's pervasive use of partial record selectors. As a result, we sometimes need to add `Default` constraints to types where they weren't already present. To guide this translation, we use the new **`set type`** edit, which allows us to change the type of a definition to a new type of our choosing. It is always safe to use this edit, as Coq's typechecker will prevent us from assigning an inconsistent type to a definition.

Type inference. Haskell's ability to perform type inference is significantly stronger than Coq's, particularly for program fragments that remain (as many do) within the bounds of Hindley-Milner type inference. For the most part, Coq's type inferencer is powerful enough, when combined with the presence of type annotations on top-level bindings, to infer all the types we need. There are, however, occasional exceptions. One subtle case is that Coq cannot infer a polymorphic type without explicit binders, as it cannot insert binders for type variables automatically.

In order to work around this, we augmented `hs-to-coq` in two ways. One is the above **`set type`** edit, which allowed us to monomorphize local functions that could have been polymorphic but were only ever used at one type. The other is that we taught `hs-to-coq` to annotate the binders

of a fixpoint with their types: to go from `let f : A -> B := fix f x := ... in ....` to `let f : A -> B := fix f (x : A) : B := ... in ....` Without this transformation, Coq's type inferencer would sometimes fail to infer the type of a fixpoint, as the type information was just too far away.

4 APPLYING EDITS TO GHC

In this section, we provide an overview and justification for the specific edits that we rely on to produce our Gallina version of GHC.

When constructing the edits that guide this translation, we follow the general design principle of "make illegal states unrepresentable"[7]. In our work, this means we set up our edits so that situations that we do not want to reason about are eliminated from the translation. As a result, even though we have axiomatized the existence of operations that work with Core `Type_`s and `Coercions`, we do not need to add axioms to our theory about the behavior of these operations.

4.1 Eliminating object-language type and coercion variables

As mentioned in Section 2.2 we use edits to eliminate the representation of type and coercion variables from GHC. As a result, our proofs need only consider those terms with no free type and coercion variables. How do we do this? We use the **`skip constructor`** edits from the previous section to change the definition of the `Var` type that represents all three forms of variables. More specifically, we eliminate the `TyVar` and `TcTyVar` constructors from `Var`, and eliminate the `CoVarId` constructor from the `IdDetails` type. The two definitions of `Var` are shown in Figure 4 for comparison.

With this modification, we would be justified in adding the following two axioms, which assert that there are no free variables to be found in types and coercions.

```
Axiom tyCoFvsOfType_empty : forall ty, TyCoRep.tyCoFVsOfType ty = FV.emptyFV.
Axiom tyCoFvsOfCo_empty   : forall co, TyCoRep.tyCoFVsOfCo co   = FV.emptyFV.
```

Of course, as these functions are defined in a module that we axiomatize (`TyCoRep`), we cannot prove or disprove this axiom in Coq. (And without this axiomatization, our **`skip constructor`** edits would fail.) We can only assume it as we work with the development.

This works, but it is more convenient to replace these axioms with edits. We can use **`rewrite`** edits to have `hs-to-coq` immediately replace these function calls with an empty set of free variables as it translates the Haskell code:

```
rewrite forall ty, TyCoRep.tyCoFVsOfType ty = FV.emptyFV
rewrite forall co, TyCoRep.tyCoFVsOfType co = FV.emptyFV
```

As a result, our proofs are simpler, as we never need to apply those axioms in our development.

This translation strategy has its advantages. In terms of proof, it is more pragmatic as the property is applied automatically. We don't need to think about it and can save our time for the details that matter. Furthermore, these edits are safer than the addition of an axiom, as – although they can be wrong – they cannot introduce unsoundness.[8] Even so, there is a cost of using a **`rewrite`** edit instead of an axiom: the rewrite leaves no trace in the generated code. If we had used the axiom approach instead, we would be able to see exactly where this sort of reasoning is required, potentially leading to more robust proofs.

4.2 Eliminating non-inductive information from the AST

We model the Core data type (Figure 2) as an *inductive* data type. However, because Haskell is a nonstrict language, this interpretation is not quite accurate. Data types are coinductive structures in

[7] Yaron Minsky, "Effective ML", https://blog.janestreet.com/effective-ml-video/

[8] Rewrites can only turn Coq terms into other Coq terms. As long as the result compiles, the result is logically consistent.

```
data Var
  = TyVar {    -- Type and kind variables
        varName    :: !Name,
        realUnique :: {-# UNPACK #-} !Int,    -- ^ Key for fast comparison
        varType    :: Kind }                  -- ^ The type or kind of the 'Var' in
    question

  | TcTyVar {                                 -- Used only during type inference
        varName        :: !Name,
        realUnique     :: {-# UNPACK #-} !Int,
        varType        :: Kind,
        tc_tv_details  :: TcTyVarDetails  }

  | Id {
        varName    :: !Name,
        realUnique :: {-# UNPACK #-} !Int,
        varType    :: Type,
        idScope    :: IdScope,
        id_details :: IdDetails,  -- Stable, doesn't change
        id_info    :: IdInfo }    -- Unstable, updated by simplifier
```

```
Inductive ... (* Var is part of a large mutually inductive type *)
with Var : Type
  := | Mk_Id (varName    : Name.Name)
             (realUnique : BinNums.N)
             (varType    : Type_)
             (idScope    : IdScope)
             (id_details : IdDetails)
             (id_info    : IdInfo)  : Var
```

Fig. 4. Haskell (top) and embedded Gallina (bottom) definitions of `Var`. The `IdDetails` data type has also been edited to eliminate the `CoVarId` data constructor, which is used to represents coercion variables. `Var` is part of a mutually defined group in Haskell, spread across several modules. In Coq, all parts of this group must be defined in one place.

Haskell. However, they are often used as though they are finite, and so the inductive interpretation of Core seems reasonable – Haskell programs are finite, after all, so they should be representable using a finite AST. And it is almost true for `Expr`.

Surprisingly, GHC treats `Expr` as a mixed inductive/coinductive data type. At first, the generated AST is finite. Then, during compilation, GHC augments identifiers with additional information about *unfoldings* (the identifier's right-hand side that may replace its occurrence) and *rules* (possible context-dependent rewrites that the optimizer can apply) [Peyton Jones et al. 2001]. The programmer can specify these optimizations through pragmas, or the compiler can create them on its own (for example, small functions tend to be inlined automatically, and GHC uses rules to specialize class methods).

This information is stored and manipulated coinductively in GHC – for instance, if the variable appears in a recursive binding, then its unfolding is an expression that may include a reference to

that same variable. At the same time, the use of coinduction is limited to this sort of metadata. For example, the GHC developers expect the `exprSize` function (which ignores information attached to identifiers) to terminate (see Figure 3).

While `hs-to-coq` could be directed to represent the `Expr` data type as a coinductive type, this would be disastrous. Coinductive data structures can only be eliminated to produce other coinductive data structures. We would not be able to translate many perfectly reasonable operations, such as `exprSize`, and we would not be able to use induction in our proofs.

Instead, to allow a fully inductive interpretation of Core, we use edits to drop all information about unfolding and rewrite rules from the data type. For rewrite rules, we replace the `RuleInfo` data type with a trivial one:

```
redefine Inductive Core.RuleInfo : Type := Core.EmptyRuleInfo.
```

We also redefine operations that work with `RuleInfo`, reflecting that our translated version of GHC is not allowed to include any information about rewriting rules.

```
redefine Definition Core.isEmptyRuleInfo : Core.RuleInfo -> bool
   := fun x => true.
```

For `Unfolding`, we use the **`skip constructor`** edit to eliminate every constructor of the data type except the no-argument `NoUnfolding` constructor.

Haskell functions that work with the Core AST also need to be edited when they use knot-tying definitions to process this coinductive data. For example, in substitution, the result that is produced when traversing a list of recursive binders is itself defined via corecursion.

```
-- | Substitute in a mutually recursive group of 'Id's
substRecBndrs :: Subst -> [Id] -> (Subst, [Id])
substRecBndrs subst bndrs
  = (new_subst, new_bndrs)
  where
    (new_subst, new_bndrs) = mapAccumL (substIdBndr (text "rec-bndr") new_subst) subst
    bndrs
```

However, the first substitution argument to `substIdBndr` is only used to update the metadata – specifically, to update rules and unfoldings, which we have made trivial. Thus, we have edited away all need to recursively use the new substitution `new_subst` above, and our translation of the `substIdBndr` function will never need it. Instead, we can pass any well-typed term in its place, and so we use the translation of the Haskell `error` function instead. (Which, like the `panic` function discussed in Section 3, is guaranteed to be sound thanks to the `Default` class.)

```
in CoreSubst.substRecBndrs rewrite forall x, CoreSubst.substIdBndr x new_subst =
  CoreSubst.substIdBndr x (GHC.Err.error Panic.someSDoc)
```

This edit allows us to produce a definition for `substRecBndrs`, which is neither recursive nor corecursive.

Why is this edit justified? First, because as previously mentioned, we have used our edits to removed all expressions that occur in the metadata. Second, because if we ever actually need to use the `new_subst` argument in our proofs, we will just find `GHC.Err.error Panic.someSDoc` instead. And since we know nothing about this substitution, we will not be able to prove much about the definition should it actually become necessary.

4.3 Replacing data structures

Parts of the code base depend on the availability of container data structures and other forms of abstract types. For example, GHC's types `VarSet`, `DVarSet`, `VarEnv`, and `DVarEnv` for variable sets and environments are implemented via the `containers` data structure `Data.IntMap`. Similarly, some strings are represented in GHC using an interned `FastString` structure.

Reasoning about GHC code requires reasoning about these data structures. For that, we need a theory. Unfortunately, although some data structures in the `containers` library have been verified [Breitner et al. 2018], `Data.IntMap` has only been translated.

Our interest is in proving the correctness of GHC, so we would rather not spend time reasoning about `Data.IntMap`. Therefore, at the beginning of this project, we axiomatized the properties of the finite map data structure that we needed for our proofs. We used this axiomatization as a foundation for a theory of the `VarSet` and `VarEnv` data types, which are used extensively in our targeted code. We also saved time by assuming that `VarSets` satisfy Coq's `FSetInterface` – this interface is a general purpose specification of finite sets that Coq's standard library builds into a rich theory. All told, we have about 3000 lines of code for reasoning about these two structures.

Since we began this project, the `containers` library has been extended. In particular, a theory of the `Data.Map` structure was added to the library. While this is not the finite map library used by GHC, it does have the same interface as `Data.IntMap`. Therefore, we were able to increase the confidence in our work by using edits to replace all uses of `Data.IntMap` with `Data.Map`, and instantiating almost all of the the axioms with the theorems about `Maps`.

Furthermore, we use data structure replacement via edits to model `DVarSet` and `DVarEnv` as well. In this case, the two modules have almost the same interface as `VarSet` and `VarEnv`; the difference between them is the order that elements are iterated over during folds, and the fold operations are never used in our modules of interest. Therefore, our edits reimplement `DVarSet` as `VarSet` and `DVarEnv` as `VarEnv`, further allowing us to avoid uninteresting verification.

4.4 Replacing computations

GHC knows two ways to calculate the set of free variables of a Core expression:

```
exprFreeVars :: CoreExpr -> VarSet
freeVars     :: CoreExpr -> CoreExprWithFVs
```

The former simply calculates the set of free variables, while the latter returns a copy of the Core expression, with all its subexpressions annotated with their free variables. From a `CoreExprWithFVs` we can get this annotation with `freeVarsOf`. The function `deAnnotate` strips the annotation.

In the proofs about the exitify transformation (which is discussed further in Section 6.3), both ways come up and we need to relate them. We could conveniently assume the following axiom:

```
Axiom freeVarsOf_freeVars: forall e, freeVarsOf (freeVars e) = exprFreeVars e.
```

Unfortunately, it is not quite true: the internal structure of the sets could differ between the two. We could assert, with a better conscience, that the two sets *denote* the same sets, but that would entail some rather tedious and unenlightening proofs that the exitify code respects this set equivalence.

Instead of adding an (unsound) axiom to our development, we can use a **`rewrite`** edit to express that, in the context of the exitify pass, the directly calculated free variables set can be used instead of the annotation:

```
rewrite forall e, freeVarsOf ann_e = exprFreeVars (deAnnotate ann_e)
```

Table 1. Translated part of GHC

	# Modules (hs)	LOC (hs)	LOC (v)
Manually defined Coq modules	—	—	418
General data structures	12	1 270	2 279
Compiler utilities	9	7 478	4 962
Core data type representation (`Core.v`)	13	8 548	5 193
Core operations and passes	15	5 268	5 452
Total	49	22 564	18 304

As an axiom, this equation would be quite bad, as in general the annotation in an `CoreExprWithFVs` could be anything. But as a rewrite edit, it is justifiable: We know exactly in which context the rewrite is applied, and we only assume that this equation holds in this particular context.

4.5 Removing dependencies

Although the Core data type and operations that we target are a small part of GHC, they have many dependencies on modules throughout the compiler. These dependencies are an issue because the translation of Haskell code to Gallina is not fully automatic. This is especially true in GHC; we have found that almost every module we have translated requires some custom edits. Because there is a cost to developing the edits necessary for the translation, we would like to do as little of it as necessary. We don't want to waste time figuring out how to translate Haskell code that we are uninterested in reasoning about.

For example, the `CoreSyn` module, which contains the AST shown in Figure 2, imports 23 different GHC-internal modules. Many of these we would also like to reason about in our formalization (e.g., `VarEnv`, `VarSet`, etc.) because they directly relate to the representation of Core terms and variables. However, this module also refers to functions and types defined in less pertinent modules including `CostCentre` (profiling information) and `Outputable` (formatting error messages). And some of these imported modules we only want to reason about abstractly. For example, we don't care how the `DynFlags` module represents compiler options, but we do need to know what the options are and how they may interact with compilation passes.

To avoid this extra complexity we make heavy use of **`skip`** and **`axiomatize`** edits. For example, of the 42 modules imported by the `DynFlags` module, only ten remain after axiomatization: five from the `base` libraries, one from the `containers` library and only four modules from GHC. As a result of these edits, our translation of the Core language draws from a total of 49 modules of GHC, which themselves contain over 22 000 lines of Haskell code. These modules are summarized in Table 1.

Another issue with defining the translation is GHC's use of recursive modules, as mentioned in Section 3. Of our 49 identified modules of interest, half of them belong to a single mutually recursive module group. While our edits allow us to merge these disparate Haskell modules into a single Gallina module, which we call `Core`, we don't want to deal with the entire module graph. Therefore we carefully axiomatize and skip definitions to successfully break the cycles (axiomatizing types and coercions is a particular help) and reduce the size of this aggregate module.

5 FORMALIZING THE SCOPING AND JOIN POINT INVARIANTS

The Core intermediate language uses a named representation for variables. GHC developers have found that working with concrete variable names, even though they require freshening to avoid capture, is the most efficient representation [Peyton Jones and Marlow 2002]. However, working with this representation, shown in Figure 4, is also subtle.

One reason for this subtlety is that identifiers in GHC store not just their `Name`, but also associated information for easy access. Some of this data never changes, such as the identifier's scope (the `IdScope`, which records whether it is a global or local identifier) or classification (the `IdDetails`, which is used to indicate specific kinds of identifiers, such as coercion variables and join points). However, the data stored in the `IdInfo` component can be updated during an optimization phase; for example, strictness analysis can use it to record how this variable might be evaluated.

For efficiency, the overloaded equality operation for variables (`==`) bases its comparison only on the unpacked `Int` stored with the variable, known as its *unique*. Despite the name, this integer is *not* guaranteed to be unique; multiple `Vars` may have the same unique but differ in their associated information. Thus, some of the properties that we specify about variables use the `almostEqual` proposition below to state that two variables are the "same"; specifically, that they differ only in their `IdInfo`, and that all other components must be identical.

```
Inductive almostEqual : Var -> Var -> Prop :=
 | AE_Id : forall n u ty ids idd id1 id2,
   almostEqual (Mk_Id n u ty ids idd id1) (Mk_Id n u ty ids idd id2).
```

5.1 Invariants about identifiers

We would like to use Coq to ensure that the Core language invariants about term variables are maintained when the AST is manipulated. First, whether a variable is a local identifier or global identifier is determined in two ways: the bits of the unique itself provide this information (and can be queried using `isLocalVar`), and it is also stored in the `idScope` component of the record (queried with the `isLocalScope` predicate).

Second is an invariant that appears in the comments in Figure 4. The integer key (i.e., the unique) used for fast comparison is actually also stored in two places in the data structure: inside the name of the variable (`varName`) as well as in the `realUnique` field (accessed via the `varUnique` function). These two values should always be in sync.

We can capture these two invariants using the `GoodVar` predicate.

```
Definition GoodVar (v : Var) : Prop :=
  isLocalVar v = isLocalScope v /\  varUnique v = nameUnique (varName v).
```

5.2 Well-scoped expressions

A well scoped expression is one where every identifier is a `GoodVar` and where all local identifiers are contained within the current `in_scope` set up to the definition of `almostEqual`. The Coq formalization of this definition is presented in Figure 5

The `VarSets` used to track the set of `in_scope` identifiers are implemented with finite maps from uniques to `Vars`. Querying whether a variable is contained within this set is done by checking whether the unique of the variable is in the domain of the map. However, that query doesn't ensure that the *same* variable is stored in the set, only one with the same unique. For our invariant, we require a stronger property: not only should the variable stored in the set have the same unique, but it should also be `almostEqual`. In that way, all of the occurrences of that variable in the expression will be forced to have the same meta-information.

Note that the `WellScoped` predicate allows shadowing in expressions. A binder can be shadowed by another binder with the same unique, but perhaps different information (name, type, etc.). The `extendVarSet` operation does not require that the key not already be present in the map; it replaces

```
Definition WellScopedVar (v : Var) (in_scope : VarSet) : Prop :=
  if isLocalVar v then
   match lookupVarSet in_scope v with
    | None => False
    | Some v' => almostEqual v v' /\ GoodVar v
    end
  else GoodVar v

Definition GoodLocalVar (v : Var) : Prop :=
  GoodVar v /\ isLocalVar v = true.

Fixpoint WellScoped (e : CoreExpr) (in_scope : VarSet) {struct e} : Prop :=
  match e with
  | Mk_Var v => WellScopedVar v in_scope
  | Lit l => True
  | App e1 e2 => WellScoped e1 in_scope /\  WellScoped e2 in_scope
  | Lam v e => GoodLocalVar v /\ WellScoped e (extendVarSet in_scope v)
  | Let bind body =>
      WellScopedBind bind in_scope /\
      WellScoped body (extendVarSetList in_scope (bindersOf bind))
  | Case scrut bndr ty alts  =>
    WellScoped scrut in_scope /\
    GoodLocalVar bndr /\
    Forall' (fun alt =>
      Forall GoodLocalVar (snd (fst alt)) /\
      let in_scope' := extendVarSetList in_scope (bndr :: snd (fst alt)) in
      WellScoped (snd alt) in_scope') alts
  | Cast e _ =>   WellScoped e in_scope
  | Mk_Type _  =>   True
  | Mk_Coercion _ => True
  end
with WellScopedBind (bind : CoreBind) (in_scope : VarSet) : Prop :=
  match bind with
  | NonRec v rhs =>
    GoodLocalVar v /\
    WellScoped rhs in_scope
  | Rec pairs =>
    Forall (fun p => GoodLocalVar (fst p)) pairs /\
    NoDup (map varUnique (map fst pairs)) /\
    Forall' (fun p => WellScoped (snd p) (extendVarSetList in_scope (map fst pairs)))
     pairs
  end.

Definition WellScopedProgram (pgm : CoreProgram) : Prop := ...
```

Fig. 5. Well-scoped Core expressions

the old value with the new one. It is an explicitly documented requirement[9] that all passes must be able to handle input that has such shadowing.

5.3 Join points

Further invariants about the use of variables in GHC arise from *join points*, one of the most recent compilation innovations in GHC [Maurer et al. 2017].

Join points are a way to express non-trivial local control flow (i.e., "jumps"). They are a more light-weight alternative to continuation-passing style. In existing paper formalizations, declaring join points and jumping to them are commonly their own syntactic categories. In GHC, the developers chose to instead represent them simply as normal let-bound function definitions and function calls, using a special marker on the function identifier (specifically, their `IdDetail` is a `JoinId`).

This leads to the following special invariants surrounding the use of variables marked as join points, quoted directly from the GHC source code:[10]

1. *All occurrences must be tail calls. Each of these tail calls must pass the same number of arguments, counting both types and values; we call this the "join arity" (to distinguish from regular arity, which only counts values).*
2. *For join arity n, the right-hand side must begin with at least n lambdas. No ticks, no casts, just lambdas! C.f.* `CoreUtils.joinRhsArity`*.*

2a. *Moreover, this same constraint applies to any unfolding of the binder. Reason: if we want to push a continuation into the RHS we must push it into the unfolding as well.*

3. *If the binding is recursive, then all other bindings in the recursive group must also be join points.*
4. *The binding's type must not be polymorphic in its return type (as defined in Note [The* `polymorphism rule of join points`*]).*

We can formalize invariants 1, 2, and 3 in our setting. However, we cannot express either invariant 2a, because we edited away the unfoldings in `IdInfo`, or invariant 4, because we have axiomatized all the type information.

In the course of doing our proofs, we found two further invariants that GHC maintains about join points:

(1) The join arity must be non-negative.
(2) A lambda-, case-, or pattern-bound variable cannot be a join point.

We omit the actual Coq definitions of the join point invariants, as they are neither informative nor elegant due to contortions to please the Coq termination checker. As before, we have a predicate that says when a complete Core program is valid:

```
Definition isJoinPointsValidProgram (pgm : CoreProgram) : Prop
```

6 REASONING ABOUT CORE

The main use that we make of the `WellScoped` and `isJoinPointsValidProgram` predicates is to show that they are invariants: that various GHC functions that work with Core programs and expressions preserve these properties.

6.1 Free variables in Core

The `exprFreeVars` operation calculates the free variables contained within a Core expression. This operation returns a `VarSet` containing those variables.

[9] In Note `[Shadowing]` in the module `CoreSyn`.
[10] In Note `[Invariants on join points]` in the module `CoreSyn`.

```
exprFreeVars :: CoreExpr -> VarSet
```

The property that we prove about `exprFreeVars` is that it is compatible with our specification of `WellScoped` expressions. The `subVarSet` operation below, defined in GHC, compares the two sets in terms of their uniques.

```
Lemma WellScoped_subset:
  forall e vs,
    WellScoped e vs -> subVarSet (exprFreeVars e) vs = true.
```

This result is more difficult to prove than it looks. For reasons of efficiency, GHC uses a special purpose data structure to store free variables gathered by `exprFreeVars` function. This data structure, `FV`, represents free variables using both a list and a set. The list is necessary to preserve the order that the free variables appear in the term; the set provides an efficient check membership check to ensure that the list does not contain duplicates. Furthermore, to make the list append operation efficient, the representation is actually a *function* that records the free variables added to the computation (a variant of a difference list [Hughes 1986]).

```
  type FV = (Var -> Bool)
              -- Used for filtering sets as we build them
            -> VarSet
              -- Locally bound variables
            -> ([Var], VarSet)
              -- List to preserve ordering and set to check for membership,
              -- so that the list doesn't have duplicates
            -> ([Var], VarSet)
```

When proving the scoping lemma we need to be able to reason about `FV`s. We define the well-formedness property of an `FV` in terms of its corresponding `VarSet`; an `FV` is well-formed if there is a `VarSet` that denotes it.

Reasoning about these sets of variables requires careful treatment of detail that can be easily ignored by a pen-and-paper proof. An example of the kind of reasoning that we need to keep track of comes from managing the different types of equalities between `VarSets`.

We use three forms of equalities to reason about `VarSets`: (1) the usual Coq equality, also known as propositional equality (denoted with `=`), which requires two `VarSets` to be structurally the same; (2) "almost" equality (denoted with `{=}`), which requires the two `VarSets` to have the same members up to the `almostEqual` relation; and (3) membership equality (denoted as `[=]`), which is the weakest equality and merely requires the two sets to contain members with the same uniques.

The formal definition of "almost" equality is given in terms of the `lookupVarSet` operation (which finds a variable by its unique):

```
Definition StrongSubset (vs1 : VarSet) (vs2: VarSet) :=
  forall var, match lookupVarSet vs1 var with
           | Some v =>  match lookupVarSet vs2 var with
                        | Some v' => almostEqual v v'
                        | None => False
                        end
           | None => True
         end.
Notation "s1 {=} s2" := (StrongSubset s1 s2 /\ StrongSubset s2 s1) (at level 70, no
    associativity).
```

The formal definition of membership equality is given in terms of the `elemVarSet` operation (which determines whether a variable is contained in the set using the unique):

```
Definition Equal (s s' : VarSet) :=
  forall a : elt, elemVarSet x s = true <-> elemVarSet x s = true.
Notation "s [=] t" := (Equal s t) (at level 70, no associativity).
```

6.2 Well-scoped substitution

GHC defines the substitution operation as the application of a `Subst` to an expression. This multi-substitution replaces multiple variables in parallel and uses separate finite maps (called substitution environments) to record the individual substitutions for identifiers, type variables, and coercion variables. In addition to these three environments, the substitution also maintains an `InScopeSet`: a set of variables that will be in scope *after* the substitution has been applied.

```
Inductive Subst : Type
  := | Mk_Subst : InScopeSet -> IdSubstEnv -> TvSubstEnv -> CvSubstEnv -> Subst.
```

The GHC substitution operation, `substExpr`, takes a `Subst` and applies it everywhere in an expression, being careful to avoid capture by renaming bound variables using the operation `substIdBndr`, shown below.

```
 Definition substIdBndr : String -> Subst -> Subst -> Id -> (Subst * Id) :=
  fun _doc rec_subst '(Mk_Subst in_scope env tvs cvs as subst) old_id =>
    let old_ty := Id.idType old_id in
    let no_type_change := orb (andb (isEmptyVarEnv tvs) (isEmptyVarEnv cvs)) true in
    let id1 := uniqAway in_scope old_id in
    let id2 := if no_type_change then id1 else
                 Id.setIdType id1 (substTy subst old_ty) in
    let mb_new_info := substIdInfo rec_subst id2 (idInfo id2) in
    let new_id := Id.maybeModifyIdInfo mb_new_info id2 in
    let no_change := id1 == old_id in
    let new_env :=
      if no_change then delVarEnv env old_id else
      extendVarEnv env old_id (Mk_Var new_id) in
    pair (Mk_Subst (extendInScopeSet in_scope new_id) new_env tvs cvs) new_id.
```

This operation takes a documentation string (`_doc`), a recursive substitution (`rec_subst`), a substitution to apply (`subst`) and the original binding variable (`old_id`) and determines whether the original binding variable needs to be renamed. In particular, this operation checks whether the identifier is already present in the `in_scope` set of the substitution (meaning that it could capture a free variable in the range of the substitution), and if so, renames the identifier using the `uniqAway` operation. Even if the variable is not captured, substitution could also change the data associated with the identifier. If the identifier was not renamed, then it is removed from the domain of the substitution (cutting off further substitution for that variable). Otherwise, the renaming is added to the domain of the substitution. In either case, the binding identifier is added to the current set of `in_scope` identifiers.

Despite this identifier shuffling, we have shown that substitution is scope-preserving. Given a well-scoped substitution and a well-scoped expression, the result of applying that substitution is also well-scoped.

```
Lemma WellScoped_substExpr : forall e s vs subst,
```

```
  WellScoped_Subst subst vs ->
  WellScoped e vs ->
  WellScoped (substExpr s subst e) (getSubstInScopeVars subst).
```

This theorem requires showing that the substitution is compatible with the current `in_scope` set for the expression. This means two things:[11]

(1) The `in_scope` set is a superset of the free variables in the range of the substitution; and
(2) The `in_scope` set is a superset of the free variables of the expression minus the domain of the substitution.

We express this constraint using the following definition, which interprets *superset* above with a strong subset relation that requires the variables of one set to be `almostEqual` to the variables contained in another.

```
Definition WellScoped_Subst  (s : Subst) (vs:VarSet) :=
  match s with
  | Mk_Subst in_scope_set subst_env _ _ =>
    minusDom vs subst_env {<=} getInScopeVars in_scope_set /\
    forall var,
      match lookupVarEnv subst_env var with
        | Some expr =>
            WellScoped expr (getInScopeVars in_scope_set)
        | None => True
        end
  end.
```

The most difficult part of proving the substitution lemma above is describing what happens when multiple binders (such as in a let expression) are potentially renamed by `substIdBndr` producing a new list of identifiers and new substitution. In this case, we needed to define the `SubstExtends` property that describes the relationship that the original substitution `s1` and list of binding variables `vars1` have to the new substitution `s2` and list of binding variables `vars2`. This relation holds when

(1) The lengths of the variable lists are the same;
(2) There are no duplicates in `vars2`;
(3) All of the vars in `vars2` are `GoodLocalVars`;
(4) The new variables `vars2` do not appear in the `in_scope` set of `s1`;
(5) The new `in_scope` set is strongly equal to the old `in_scope` set extended with the new variables;
(6) The `in_scope` set, with the addition of the old variables, minus the domain of the new environment, is a subset of the new `in_scope` set; and
(7) Anything in the new environment is either a renamed variable from the old environment or was already present.

These conditions are not present in the GHC source code, but are necessary to prove the well-scopedness property above. Together, they ensure that when multiple binders are renamed simultaneously, the invariants about binding are still preserved.

6.3 Exitification

When join points were added to GHC, they opened the door for new program transformations and simplifications [Maurer et al. 2017]. One such opportunity is the ability to float a definition into a lambda.

[11] These conditions are currently expressed in a comment in the GHC source code. However, the comments about substitution invariants in version 8.4.3 were out of date and updated by the GHC developers after correspondence with the authors.

Consider first the situation with regular functions, as in code like this:

```
let t = foo bar in
let f x = t (x*x) in
body mentioning f
```

It might be beneficial to inline `t`, replacing its occurrence in `f` with `foo bar`, as this can create new optimization opportunities in the body of `f`.

However, in general the compiler cannot do that in situations like this. As the code stands, `t` is evaluated at most once. If it was inlined into `f`, it would instead be evaluated as often as `f` is called. Thus, if `t` is expensive to compute, this could be an arbitrarily bad pessimization, and so GHC does not inline `t`.[12]

The story is suddenly much simpler if `f` is not a general function, but actually a *non-recursive* join point `j_f` (we indicate join points with names starting with `j_`):

```
let t = foo bar in
let j_f x = t (x*x) in
body mentioning j_f
```

The join point invariants guarantee that all calls to `j_f` in the body are in tail-call positions. This implies that `j_f` is called at most once (more precisely: jumped to at most once), and the compiler may inline `t` at will.

This does not hold for recursive join points:

```
let t = foo bar in
let j_go n x y =
        if n = 0
        then t (x*x)
        else j_go (n-1) (x*x) (x+y)
in body mentioning j_go
```

Because `j_go` is *recursive*, its right-hand side will likely be evaluated multiple times, so inlining `t` would again risk repeated evaluation of `t`.

Or would it? More careful inspection of the code above reveals that in the case where `t` is evaluated, no further recursive call to `j_go` occur. Or put differently: `t` is on the *exit path* of the loop represented by `j_go`, and inlining is safe after all.

The purpose of the exitification optimization is to find and recognize situations like these, and to transform the code so that it is obvious to the general purpose simplifier that `t` is used at most once. It does so by floating the expression on the exit path out of the recursive loop, into a non-recursive join point:

```
let t = foo bar in
let j_exit x = t (x*x) in
let j_go n x y =
        if n = 0
        then j_exit x
        else j_go (n-1) (x*x) (x+y)
in body mentioning j_go
```

[12] It *would* be safe to inline `t`, however, if the compiler knows that that `f` is called at most once. And indeed there are multiple elaborate analyses in GHC that try to answer the question of whether `f` is called more than once [Breitner 2018a; Sergey et al. 2017].

Now we are again in the same situation we were with the non-recursive `j_f` before, and the simplifier will be able to inline `t`.

Theorems proved. Any transformation that moves code from one scope to another needs to be careful about preserving the well-scopedness invariants, and exitification is no exception. It must delicately juggle names and scopes: the newly created exit join points must not shadow existing names, and they need a parameter for each free variable that is no longer in scope outside the recursive join point (`x` in this case, but not `y`). This made the proof that that exitification preserves well-scopedness tricky.

Given that exitification only makes sense in the context of join points, and that it creates new join points, we were also naturally interested in knowing that the code that exitification produces adheres to the invariants about join points; for example, that `j_exit` has the right number of lambdas and is invoked only from tail-call positions.

Furthermore, it turns out that the exitification code is actually not well-defined on Core terms that violate the join point invariants, so they also became a precondition for the well-scopedness proofs.

Therefore, we proved a combined theorem:

```
Theorem exitifyProgram_WellScoped_JPV: forall pgm,
  WellScopedProgram pgm -> isJoinPointsValidProgram pgm ->
  WellScopedProgram (exitifyProgram pgm) /\ isJoinPointsValidProgram (exitifyProgram
    pgm).
```

A bug is found. While trying to show that exitification preserves well-scopedness, we found that it did not, at least in some obscure situations. Consider the following opportunity for floating out the expression `foo x`:

```
let j_go (x :: Bool) (x :: Int) =
        if x == 0
        then foo x
        else j_go True (x - 1)
in ...
```

Note that the second parameter of `j_go` shadows the first. The previous version of exitification kept a list of all locally bound variables, in their binding order, and then abstracted the exit expression over this list of variables:

```
let j_exit (x :: Bool) (x :: Int) = foo x in
let j_go (x :: Bool) (x :: Int) =
        if x == 0
        then j_exit x x
        else j_go True (x - 1)
in ...
```

But this is now wrong: The first argument to `j_exit` should be of type `Bool`, but a value of type `Int` is used instead!

The impact of this bug was relatively low, as under normal circumstances, the Core passed to the exitification pass does not exhibit such shadowing. Nevertheless, it is a bug that was worth fixing.[13] In particular, users of GHC can insert custom Core-to-Core passes at any point in the pipeline via compiler plugins, and such plugins are becoming more popular [Bolingbroke 2008;

[13] GHC issue #15110, fix first released with GHC 8.6.1.

Breitner 2018b; Elliott 2017; Farmer et al. 2012; Grebe et al. 2017; Lippmeier et al. 2013]. As plugins are less constrained, GHC must be able to handle any possible shadowing in Core in case such passes introduce them.

Because of this bug, the version of the `Exitify` module that we current translate and verify is the version from GHC HEAD, not from GHC 8.4.3.

7 A MECHANIZED FORMALIZATION GAP

There are two forms of formalization gap in our work: the gap introduced by the translation itself, that lies in the difference in semantics between the Haskell and Gallina versions of GHC; and the gap that derives from the introduction of axioms in our proofs.

Pragmatically, we cannot work without either. Because Haskell and Coq do not have the same semantics, and because GHC takes advantage of Haskell idioms that are difficult to map into Coq, we will always have some sort of translation gap. The second form of gap is also important in terms of pragmatic proof development. We want to reason about the most interesting parts of the code base first and defer the verification of other parts, perhaps indefinitely.

Note that it is specifically *not* our goal to verify everything: there is just too much GHC. Instead, we would like to reason about the system given appropriate assumptions. In the next subsection, we justify the axioms that we rely on as part of this proof development.

7.1 Axioms

Overall, because of careful choices in the design of our edits, as discussed in Section 4, we require few axioms when reasoning about the translated code. In particular, the axioms that we do require relate directly to the objects of our case study (i.e., to variables and variable sets).

Uniques and `uniqAway`. For example, one operation that we axiomatize is the `uniqAway` operation used in the substitution and exitify operations to produce fresh variable names.

```
uniqAway :: InScopeSet -> Var -> Var
```

In GHC, this operation tries to find a fresh variable (for a given `InScopeSet`) by repeatedly guessing. If it cannot find one after one thousand guesses, the operation gives up and `panic`s. The correctness of substitution and exitify depends on the `uniqAway` operation always successfully producing fresh variables. However, we cannot prove the totality of this operation from the implementation in GHC: there is no guarantee that one thousand guesses will be enough.

Therefore, we ignore its (translatable) implementation and axiomatize it along with the properties that we require. In particular, we rely on axioms that state that `uniqAway`:

- returns a variable that is not in scope,
- preserves the invariants of good variables,
- preserves any information associated with the variables, and
- doesn't modify variables that don't need to be freshened.

`VarSets`. `VarSets` are implemented by finite maps in GHC. A set of variables is a finite map from a variable's unique (i.e., an integer) to the variable itself. As described in Section 4.3, we use edits to implement `VarSets` with a finite map implementation that comes with a rich set of properties.

However, one significant set of axioms that we require about this implementation are those that justify that a `VarSet` satisfies Coq's notion of a finite set, the `FSetInterface`. Because this signature comes from the Coq standard library,[14] we can assume it holds of this data structure with little

[14] https://coq.inria.fr/library/Coq.FSets.FSetInterface.html

concern. Filling in the details of this instantiation would take some effort, but is unlikely to be problematic.

However, there is one axiom that we assume about `VarSet`s that does not directly follow from a it being an implementation of a finite set. This invariant is recorded by a comment in the GHC source code about the `lookupVarSet` function:

```
lookupVarSet :: VarSet -> Var -> Maybe Var
                -- Returns the set element, which may be
                -- (==) to the argument, but not the same as
```

In other words, if a `VarSet` maps a unique to a `Var`, then that key must be the same unique as the one stored in the `Var`.

We need this property in our proofs. Therefore, we add an axiom that states that all `VarSet`s satisfy this property.

```
Definition ValidVarSet (vs : VarSet) : Prop :=
  forall v1 v2, lookupVarSet vs v1 = Some v2 -> (v1 == v2).
Axiom ValidVarSet_Axiom : forall vs, ValidVarSet vs.
```

Why is this axiom justified? This property is an invariant of the implementation. In particular, in defining the `VarSet` type, GHC uses an abstract data type called a `UniqFM` to hide the finite map implementation. This `UniqFM` automatically accesses the unique from any element to use as its key when it is added to the data structure. Therefore, it is impossible to create a `VarSet` where this invariant does not hold.

One way to avoid this axiom in a Coq development would be to implement the `UniqFM` type using a sigma type of a finite map with the invariant above. All operations on this type would then need to prove that they preserve this invariant.

However, we have not added this reasoning to our code, as it is beyond the scope of `hs-to-coq`. In particular, this modification changes the signature of some of the `UniqFM` operations: because the invariant talks about the Unique key stored in the value, simply stating the invariant requires a `Uniqueable` class constraint. As a result, operations such as `emptyUFM` that do not require this constraint in Haskell, would need this constraint in Gallina. Exploring how to automatically support such a translation with `hs-to-coq` would be interesting future work.

Variable scopes. Our definition of `GoodVar` ensures that two parts of the variable remain in sync: some bits of the unique of the variable that indicate whether it is a local or global variable, and one part of the `Var` type that includes this same information.

Our proofs about the `WellScoped` predicate (which requires this property for all term variables appearing in the expression) prove that operations preserve this invariant. But our system also subtly assumes this property, too. In particular, in our translation, we redefine the `isLocalVar` function. In GHC, this function looks at the `idScope` in the `Var` to determine whether a variable is a local var. However, in this work, we redefine this function so that it makes the decision based on the unique instead. As long as these two components stay in sync, this change does not affect the behavior of the function.

Despite this complexity, the value of making this edit is that we can use the fact that `isLocalVar` respects GHC's definition of `(==)` for variables when reasoning about the free variable function.

7.2 Mechanization

Although our edits introduce a formalization gap in our translation, this gap is made explicit as an artifact of our development, in much the same way as the axioms above. We call both these

explicit assumptions together a *mechanized formalization gap* – a machine-readable record, in the `hs-to-coq` edit files and axioms, of all the simplifying assumptions we have made.

We see benefit in constructing such artifacts. Past work on mechanical verification for GHC has focused on verifying descriptions of parts of the compiler developed by hand [Breitner 2015a,b]. These models simplify and elide many details that appear in the implementation. How do we know what simplifications have been made? How do we know how well these models correspond to the code that GHC implements?

In contrast, by using edits and axioms, we have the following benefits from a mechanized formalization gap:

- *Provenance*: The model is directly and observably connected to the actual implementation.
- *Richness*: The model can include more detail because it was developed by eliminating the parts that turn out to be too difficult to reason about, as opposed to only adding what was *a priori* considered to be important.
- *Mechanical assistance*: Developing a consistent mathematical model of a code base as huge and intertwined as GHC is too difficult to do by hand at the scale we are working at.

Furthermore, this process does not eliminate the possibility of the development of more precise characterizations of the compiler implementation in future work. Indeed, the recorded simplifications could be used to express the refinement relation with a future, more detailed model.

8 RELATED WORK

Liquid Haskell. Liquid Haskell [Vazou 2016] also shares the goal of verifying real-world Haskell code. Breitner et al. [2018] extensively discusses the relation between Liquid Haskell and prior uses of `hs-to-coq` as well as comparisons to other tools and methodologies for verifying Haskell programs. Our work extends the scale of verification relative to these systems, demonstrating that mechanical reasoning through shallow embeddings is possible for code extracted from a system with more than a hundred thousand lines of code.

Refinement. Refinement relations, along with forward and backward simulation relations that generalize them, are commonly used to describe a correspondence between two programs doing the same "important" computations [Abadi and Lamport 1991; Lynch and Vaandrager 1995; Milner 1971]. For example, seL4 uses refinements to connect kernel code written in C with compiled binary code, and to extend verification made on the former to the latter [Sewell et al. 2013]. CompCert uses backward simulation to express its semantic preservation theorem [Leroy 2009].

Refinement relations are also commonly used to establish a correspondence between a concrete implementation and a specification for which formal analysis can be more easily performed. For example, seL4 defines an abstract model of its operating system and bases its verification on this. Refinements then show that all the Hoare logic properties of the abstract model also hold for the kernel source code [Klein et al. 2014]. CertiKOS also uses simulations to construct certified abstract layers to facilitate modular reasoning [Gu et al. 2015]. The usage of refinement relations is also popular for reasoning about concurrent or distributed systems [Gu et al. 2018; Hawblitzel et al. 2015a,b; Koh et al. 2019; Lamport 1983] and it has been shown that observational refinement is equivalent to linearizability, a popular correctness condition of concurrent systems [Filipovic et al. 2009].

Refinement has also been used from the other direction in program development: the process starts from a high-level specification and then applies several refinement steps to derive a concrete implementation [Cohen et al. 2013; Delaware et al. 2015; Dénès et al. 2012; Wirth 1971]. A recent success story of this approach is a high performance implementation of an elliptic-curve library that has been deployed to Chrome, Android, and CloudFlare [Erbsen et al. 2019].

There is a superficial similarity between refinement and this work. In both cases there is a connection between two versions of a system, where one form is more suitable for mechanical verification. However, whereas refinement proves the equivalence between the two systems so that the proofs directly carry over, our work merely establishes a relationship (the edit-based translation) and does not attempt to reason about it. Furthermore, with the exception of synthesis (e.g., Fiat [Delaware et al. 2015]), with refinement the two systems are constructed by hand, while we use an automated transformation.

Prior work on verifying compilers. Compilers play critical roles in any software development, and therefore it is important to ensure that they are correct. Despite decades of effort invested in testing compilers, we are still far from the ideal. Yang et al. [2011] have used CSmith, a randomized test-case generation tool, to exhibit many bugs in mainstream compilers including GCC and LLVM. In the darkness of software bugs, formal verification has offered new hope. A ground-breaking development in this direction was CompCert, a formally verified optimizing C compiler implemented in Coq [Leroy 2006]. The same study by Yang et al. fails to find any wrong-code errors in CompCert despite devoting significant resource to the task. Other inspiring endeavors in this direction include Vellvm, which formalizes the operational semantics of LLVM IR and its SSA-based optimizations in Coq [Zhao et al. 2012, 2013]; CakeML, a formally verified ML compiler implemented in HOL4 [Tan et al. 2019]; and CertiCoq, a verified optimizing compiler for Coq [Anand et al. 2017].

Our work differs from these impressive results because we target the actual source code of GHC, a mainstream, industrial-strength compiler that was written without verification in mind. The advantage is that the verification is separate from the development: the compiler developers don't have to know formal reasoning techniques, and verification does not interact with their workflow. In contrast, these aforementioned systems are based on new implementations that were developed in conjunction with the verification of their properties. While we agree that this approach is more likely to lead to completely verified software system, it does not enable partial verification or mechanically-assisted reasoning for existing code.

Prior work on variable binding. Our proofs in GHC are mainly about the use of variables in the abstract syntax of lambda-calculus based intermediate languages. Many mechanized developments of lambda terms select variable representations, such as de Bruijn indices [Schäfer et al. 2015], locally nameless representation [McKinna and Pollack 1999], higher-order abstract syntax [Chlipala 2008; Despeyroux et al. 1995], or nominal logic [Urban and Tasson 2005], that simplify this reasoning. Of the solutions presented to the POPLmark challenge [Aydemir et al. 2005] only one (by Aaron Stump) used a fully named representation of binding like GHC does.

The representation of variables in verified compilers often varies based on the phase. CakeML uses strings for variables at the source level but uses closures instead of substitution to describe their semantics. The compiler then translates this representation to an intermediate language that uses de Bruijn indices [Kumar et al. 2014; Tan et al. 2019]. CertiCoq works in the opposite way, relying on a de Bruijn representation for the AST of Gallina, and then translating to a named representation for the CPS conversion stage [Anand et al. 2017].

9 FUTURE WORK AND CONCLUSIONS

There is of course, much more to verify about GHC. Indeed, our work is only the start of reasoning about even the term variable invariants in the Core intermediate language.

In addition to the passes that we have considered in this paper, we have also translated a few other Core optimization passes, including those for common subexpression elimination and call-arity optimization. However, several of the GHC optimization passes are still untranslatable, due to heavy use of mutual recursion in their definitions. The edits discussed in Section 3 are not

sufficient to show the that functions are terminating, yet alternative approaches such as deferred termination [Breitner et al. 2018] are not available for mutual recursion. In future work, we hope to extend the capabilities of `hs-to-coq` with respect to this issue. In particular, extending `hs-to-coq` with support for axiomatizing the behavior of mutually recursive functions [Bove and Capretta 2005] or integrating it with the Equations package [Mangin and Sozeau 2018] might provide a more general approach to this issue.

As discussed in Section 3, our edits remove information about rules and unfolding information from Core AST terms. This works well for passes, such as exitify, that do not care about this information, but what about those that do? What if we wanted to reason about this data? Or what if we wanted to reason about other data that is coinductively represented in GHC? One approach would be to augment the `hs-to-coq` edit language so that it can transform coinductive representations to inductive models, perhaps by storing this information elsewhere in the AST or in additional arguments to compiler passes.

In future work, we would like both to verify more optimization passes and to check more invariants about the Core data structures. In particular, the abstract syntax tree used for GHC's intermediate Core language (Figure 2) is very simple, but this isn't the whole story: GHC maintains and relies on many more invariants of this data structure besides the scoping and join point invariants considered here.

In particular, there are additional structural invariants to reason about, such as that in a `case` expression, a default case must come first and the other cases must be ordered. More significantly, Core is a typed language, and we could also verify that Core passes preserve the typing of Core terms. The Core type system includes complicated rules like the *let/app invariant*, which govern when the right-hand side of a `Let` or the argument in an application can be of unlifted types. Furthermore, connecting our manually written invariants to the Core type checker (`CoreLint`) would be interesting future work.

Finally, we would also like to reason not just about Core invariants but about the semantics for Core terms. It is not difficult to define a simple call-by-name semantics for this AST, which would let us argue that operations are semantics preserving.

In the end, although our efforts are tailored to GHC and its invariants, they reflect the general challenges and benefits of bringing interactive verification to large scale software systems. In particular, our work has demonstrated that shallow embeddings are available and useful even at the scale of something like GHC. Although our edit files simplify some aspects of the system, enough details remain to benefit from mechanical reasoning.

Given the interactive nature of tools like Coq proof assistant, we doubt that this approach will scale to the verification of *complete* systems. However, this work has demonstrated, through its extended case study that it is possible to take an existing system and reason about part of it. We hope that this work will lead to further use of mechanical reasoning in other large software projects.

ACKNOWLEDGMENTS

Thanks to Leonidas Lampropoulos and William Mansky for their valuable comments on a draft of this paper. Thanks to Josh Cohen, Christine Rizkallah, and John Wiegley for assistance with the development of `hs-to-coq` and the theory of the `containers` library. This material is based upon work supported by the National Science Foundation under Grant No. 1521539.

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The POPLmark Challenge. In *The 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Oxford, UK, 50–65.

Max Bolingbroke. 2008. Compiler Development Made Easy. *The Monad. Reader Issue 12* (2008).

Ana Bove and Venanzio Capretta. 2005. Modelling General Recursion in Type Theory. *Mathematical. Structures in Comp. Sci.* 15, 4 (Aug. 2005), 671–708. https://doi.org/10.1017/S0960129505004822

Joachim Breitner. 2015a. Formally proving a compiler transformation safe. In *Haskell Symposium*. ACM. https://doi.org/10.1145/2804302.2804312

Joachim Breitner. 2015b. The Safety of Call Arity. *Archive of Formal Proofs* (Feb. 2015). http://afp.sf.net/entries/Call_Arity.shtml

Joachim Breitner. 2018a. Call Arity. *Computer Languages, Systems & Structures* 52 (2018). https://doi.org/10.1016/j.cl.2017.03.001

Joachim Breitner. 2018b. A promise checked is a promise kept: inspection testing. In *Haskell Symposium*, Nicolas Wu (Ed.). ACM. https://doi.org/10.1145/3242744.3242748

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe zero-cost coercions for Haskell. *Journal of Functional Programming* 26 (2016). https://doi.org/10.1017/S0956796816000150

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying Hs-to-coq to Real-world Haskell Code (Experience Report). *Proc. ACM Program. Lang.* 2, ICFP, Article 89 (July 2018), 16 pages. https://doi.org/10.1145/3236784

Adam Chlipala. 2008. Parametric Higher-order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, New York, NY, USA, 143–156. https://doi.org/10.1145/1411204.1411226

Cyril Cohen, Maxime Dénès, and Anders Mörtberg. 2013. Refinements for Free!. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. 147–162. https://doi.org/10.1007/978-3-319-03545-1_10

The Coq development team. 2004. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.0.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 689–700. https://doi.org/10.1145/2676726.2677006

Maxime Dénès, Anders Mörtberg, and Vincent Siles. 2012. A Refinement-Based Approach to Computational Algebra in Coq. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*. 83–98. https://doi.org/10.1007/978-3-642-32347-8_7

Joëlle Despeyroux, Amy Felty, and André Hirschowitz. 1995. Higher-order abstract syntax in Coq. In *Typed Lambda Calculi and Applications*, Mariangiola Dezani-Ciancaglini and Gordon Plotkin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–138.

Richard Eisenberg. 2015. System FC, as implemented in GHC. https://github.com/ghc/ghc/blob/master/docs/core-spec/core-spec.pdf

Conal Elliott. 2017. Compiling to categories. *PACMPL* 1, ICFP (2017), 27:1–27:27. https://doi.org/10.1145/3110271

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*.

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. 2012. The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs. In *Haskell Symposium*. ACM. https://doi.org/10.1145/2364506.2364508

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2009. Abstraction for Concurrent Objects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. 252–266. https://doi.org/10.1007/978-3-642-00590-9_19

Mark Grebe, David Young, and Andy Gill. 2017. Rewriting a shallow DSL using a GHC compiler extension. In *GPCE*. ACM. https://doi.org/10.1145/3136040.3136048

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*.

595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 646–661. https://doi.org/10.1145/3192366.3192381

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015a. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 1–17. https://doi.org/10.1145/2815400.2815428

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015b. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. 449–465. https://doi.org/10.1007/978-3-319-21668-3_26

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

John Hughes. 1986. A Novel Representation of Lists and its Application to the Function "reverse". *Inf. Process. Lett.* 22, 3 (1986), 141–144. https://doi.org/10.1016/0020-0190(86)90059-1

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* 32, 1 (2014), 2:1–2:70. https://doi.org/10.1145/2560537

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. 234–248. https://doi.org/10.1145/3293880.3294106

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/2535838.2535841

Leslie Lamport. 1983. Specifying Concurrent Program Modules. *ACM Trans. Program. Lang. Syst.* 5, 2 (1983), 190–222. https://doi.org/10.1145/69624.357207

Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/1111037.1111042

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Ben Lippmeier, Manuel M. T. Chakravarty, Gabriele Keller, and Amos Robinson. 2013. Data flow fusion with series expressions in Haskell. In *Haskell Symposium*. ACM.

Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233. https://doi.org/10.1006/inco.1995.1134

Cyprien Mangin and Matthieu Sozeau. 2018. Equations reloaded. (July 2018). https://hal.inria.fr/hal-01671777 working paper or preprint.

Simon Marlow and Simon Peyton Jones. 2012. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications, Volume II*, Amy Brown and Greg Wilson (Eds.). Lulu, Chapter 5.

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 482–494. https://doi.org/10.1145/3062341.3062380

James McKinna and Robert Pollack. 1999. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning* 23, 3 (01 Nov 1999), 373–409. https://doi.org/10.1023/A:1006294005493

Robin Milner. 1971. An Algebraic Definition of Simulation Between Programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*. 481–489. http://ijcai.org/Proceedings/71/Papers/044.pdf

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler Inliner. *Journal of Functional Programming* 12, 5 (2002). https://doi.org/10.1017/S0956796802004331

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015 (LNAI)*, Xingyuan

Zhang and Christian Urban (Eds.). Springer-Verlag.

Ilya Sergey, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Joachim Breitner. 2017. Modular, higher order cardinality analysis in theory and practice. *Journal of Functional Programming* 27 (2017). https://doi.org/10.1017/S0956796817000016

Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 471–482. https://doi.org/10.1145/2491956.2462183

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18)*. ACM. https://doi.org/10.1145/3167092 New York, NY, USA.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Types in Languages Design and Implementation (TLDI)*. ACM. https://doi.org/10.1145/1190315.1190324

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *J. Funct. Program.* 29 (2019), e2. https://doi.org/10.1017/S0956796818000229

Christian Urban and Christine Tasson. 2005. Nominal Techniques in Isabelle/HOL. In *Proceedings of the 20th International Conference on Automated Deduction (CADE' 20)*. Springer-Verlag, Berlin, Heidelberg, 38–53. https://doi.org/10.1007/11532231_4

Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph.D. Dissertation. University of California, San Diego, USA. http://www.escholarship.org/uc/item/8dm057ws

Niklaus Wirth. 1971. Program Development by Stepwise Refinement. *Commun. ACM* 14, 4 (1971), 221–227. https://doi.org/10.1145/362575.362577

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 283–294. https://doi.org/10.1145/1993498.1993532

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 427–440. https://doi.org/10.1145/2103656.2103709

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2013. Formal verification of SSA-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 175–186. https://doi.org/10.1145/2491956.2462164