Verification of System FC in Coq

Dept. of CIS - Senior Design 2014-2015*

Tiernan Garsys
tgarsys@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Tayler Mandel
tmandel@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Lucas Peña
lpena@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Noam Zilberstein
noamz@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

ABSTRACT

Haskell is a statically-typed functional programming language that is commonly used for the robust compile-time guarantees provided by its type system. Despite this usage, the type safety of Haskell has not been mechanically proven; it may be possible to write Haskell programs that type-check at compile-time but fall into an inconsistent state at runtime. Many safety critical systems such as flight controllers, self driving cars, and missile controllers are powered by software. If the type systems underlying this code are not verified, then the code itself is unsafe.

This work approaches this problem by verifying System FC, the formalization of the core language of the Haskell compiler. It presents a mechanized proof for a large subset of System FC using the Coq Proof Assistant. The type system of System FC is the basis for the type system of Haskell, so the results translate directly into safety guarantees at the program level.

1. INTRODUCTION

Haskell is often used for its strong, static type system. This system allows for programmers to specify types for their programs that provide some guarantees about the behavior of the program before it is ever executed. Haskell is often thought to be a type safe language, meaning any program accepted as well-typed will never reach an inconsistent state [4]. This type safety has never been mechanically proven for Haskell, and the existence of such a proof would rule out a whole class of potential errors in Haskell code. This mechanical proof could also allow for more general adoption of the Haskell language for safety critical applications.

A fully mechanical proof is also easily extensible so that additional properties could be proven. A mechanized proof of System FC (a language specification implemented by the Haskell compiler) would provide a starting point for authors of Haskell language extensions or optimizations to formally prove the type safety of their extensions. If all extensions and optimizations were proven within a working proof system, it would be easier for language designers to reason about the implications of their addition to the base language that is being extended.

Proving Haskell's type safety is done through a formalization and proof of a specification of Haskell's core language, GHC Core. GHC Core is first intermediate representation in the Haskell compilation chain. This language is more explicit about the types of expressions and is much easier for the compiler to reason about than normal Haskell code. GHC Core is an implementation of a language specification called System FC. The system described is a formalization and proof of the type safety of System FC. Using the Coq Proof Assistant, the syntax, typing relations, and evaluation rules of System FC are first formalized, and then proofs of several properties are completed. Most importantly, the Soundness theorem is proven as an application of the Progress and Preservation theorems. The Progress theorem states that no well-typed program is in a state where further evaluation is undefined, and the Preservation theorem states that if a program can take a discrete step toward its completion, the type of the expression will not change as a result of taking the step. From these, Soundness states that no well-typed program will ever reach a stuck state during execution.

In the creation of this formalization and proof, there are several technical challenges. A language is often described with variables represented as strings. These strings could be the same and could allow for shadowing and troubles when substituting variables within expressions. A formalization without taking this into account would allow for substituting variables to change the semantics of a program. As a means of avoiding naming conflicts, variables are described using De Bruijn indices [7] instead of strings.

The resulting system is a full formalization and mechanical proof of the type saftey of System F [3] with coercions [8]. This is the first mechanical proof of this particular subset of System FC, and it can be used as a base formalization and proof system for future work in proving the type safety of Haskell as well as any language extensions or optimizations in GHC.

*Advisors: Stephanie Weirich (sweirich@cis.upenn.edu), Richard Eisenberg (eir@cis.upenn.edu).

2. BACKGROUND

2.1 Haskell

Haskell is a functional programming language originally released in 1990. Haskell is a statically-typed language; the types of all expressions and variables are determined at compile-time via either type inference or explicit type annotations by the programmer. Haskell is strongly-typed, meaning that the usage of values in identifier declarations and functions must be consistent with the statically-declared types of these functions; Haskell will not implicitly convert values from one type to another in order to satisfy the static typing of the program. Haskell is also a purely-functional language; a function defined in Haskell is guaranteed not to have side-effects such as I/O or the mutation of data structures in the running environment unless this is explictly allowed by the programmer.

The strong static typing of Haskell is attractive because it allows the programmer to make certain guarantees about the properties of his or her program at compilation, rather than runtime. Such guarantees include the guarantee that a function is never called with an invalid input type, or the guarantee that a function defines a behavior for null inputs. By determining these properties at compile-time, one can rule out entire classes of errors prior to ever running the program.

These advantages are founded on the believed type safety of Haskell. Type safety in a programming language refers to its resilience to type errors at runtime. In the context of a statically-typed language such as Haskell, type safety requires ensuring that the guarantees of behavior made at compile-time are preserved during program execution. Given type safety, one can be sure that the program does not exhibit undefined behavior (such as segmentation faults) during execution.

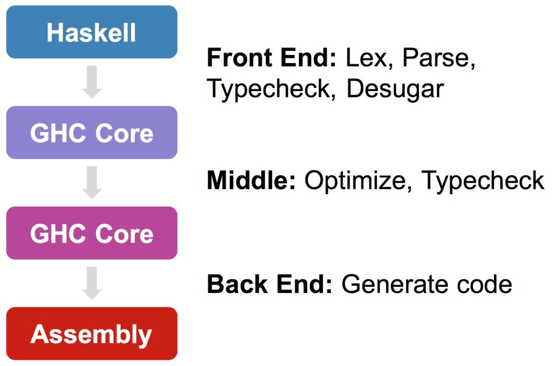
2.2 GHC Core

The Glasgow Haskell Compiler, or GHC, is an optimizing compiler used to generate native executables from Haskell code. As with most compilers, GHC compiles programs in multiple phases, translating the source between various intermediate representations. These phases of compilation are outlined in Figure 1. Whereas a surface language, such as Haskell, is structured to be easy for humans to work with, these intermediate representations (also known as *core languages*) are designed to be easy for a compiler to modify and optimize.

In the first phase of compilation, Haskell code is typechecked and then converted into a desugared intermediate language called GHC Core. The conversion from Haskell to GHC Core involves adding explicit type annotations to all values (which, in Haskell, can be omitted by the programmer and subsequently inferred when type-checking), adding explicit type parameters to type declarations with polymorphic types, and assigning all identifiers globally-unique names. Once converted, GHC performs optimizations on the resulting GHC Core code before passing the program on to later stages of compilation.

2.3 System FC

GHC Core is an implementation of System FC, a formal language specification designed for use as an intermediate representation during the compilation of Haskell code [5].



Haskell
GHC Core
GHC Core
Assembly
Front End: Lex, Parse, Typecheck, Desugar
Middle: Optimize, Typecheck
Back End: Generate code

Figure 1: The GHC compilation process

The theoretical foundation of System FC is the untyped lambda calculus, a formal language used to express computations [4]. The untyped lambda calculus builds up computations in terms of functions, allowing for one to construct function abstractions (which define a function in terms of some input variable) and function applications (which take an input argument and substitute instances of the function's input variable with that argument). From its small core, one can extend the untyped lambda calculus with additional features which may be useful in studying features of a language implementing an equivalent type system. One of the most widely-known extensions to the untyped lambda calculus is the simply-typed lambda calculus. This system is similar to the untyped lambda calculus, but adds the annotation of types to the arguments of functions [3]. Whereas any value may be passed as an argument to a function in the untyped lambda calculus, an expression is only valid in the simply-typed lambda calculus if the type of an expression passed as an argument to a function is the same as the type annotation for that variable. This typing system is further extended by System F, also known as the polymorphic lambda calculus [3]. In this system, the direct precursor to System FC, the simply-typed lambda calculus is extended with the ability to express universally quantified types. Universally quantified types are types which are parametrized by other types: for example, System F allows one to express a type $\Lambda\alpha.\alpha \rightarrow \alpha$, which is the type of a function which takes an input type α and a single expression of type α and returns an expression of type α. This feature essentially allows for functions to take types as parameters, granting the ability to define functions whose actual types vary based on these input types, which can be useful in modeling surface language features such as parametric polymorphism or generics [3].

System FC is a programming language specification defined in 2007 as a superset of the features present in System F [5]. System FC was introduced due to perceived shortcomings in System F as an intermediate language specification in GHC. While System F used to be sufficient as an intermediate representation for Haskell code, language designers have started experimenting with novel type systems such as generalized algebraic datatypes (or GADTs) and associated types that are either difficult or impossible to represent in System F. System FC ameliorates these issues by introducing features such as type coercions, type families, and datatypes. Type coercions allow for type families and generalized algebraic datatypes to exist in System FC by acting as a witness for equality between syntactically different types [8]. Types can be equal in different ways and therefore there is a complex set of coercion rules that can be used to construct correct equality proofs [1]. These equality proofs are responsible for most of the power of System FC over System F. They allow a conversion from one type to another. A basic example of the usefulness of coercions is provided through an example piece of code in Haskell in Figure 2 and its translation into System FC in Figure 3.

```
data G a where
  G1 :: G Int
  G2 :: G Bool

f :: G a -> a
f G1 = 5
f G2 = True
```

Figure 2: Coercion in Haskell

```
G :: ⋆ → ⋆
G1 :: ∀ (a :: ⋆). a ∼ Int → G a
G2 :: ∀ (a :: ⋆). a ∼ Bool → G a

f :: ∀ (a :: ⋆). G a → a
f = λ(a :: ⋆). λ(x :: G a)
   case x of
     G1 c → 5 ▷ sym c
     G2 c → True ▷ sym c
```

Figure 3: Coercion in System FC

This is a basic example of datatypes in Haskell and a trivial use of them. `G` is a parameterized datatype with kind $\star \to \star$. A *kind* in Haskell represents the type of a type constructor. In Haskell, the types Int and Bool both have kind $\star$. `f` is a function that takes something of type `G a` and returns something of type `a`. In Haskell, this only compiles because of coercions.

Here, `G1` is stating that for all types `a` of kind $\star$, if `a` can be coerced to an Int, then one can obtain something of type `G a`. `G2` is defined similarly. Now, in the `G1` case, in order for the function `f` to correctly yield something of type `a`, `5` needs to be coerced to be of type `a`. This is accomplished using the rule from the construction of `G1`, since `G1` requires that an `a` can be coerced to an `Int`. The symmetry of this rule allows for `5` to be coerced to an `a`, which is required in the body of `f`.

2.4 Type Safety

Types are used in programming languages as a means of classifying pieces of data; common examples of types include integers, characters, and boolean values. Types are used to determine guidelines for how one may work with a piece of data, such as the range of values that the data may take or the operations that are allowed to be performed with a given piece of data. By explicitly forming rules about how one may work with various pieces of data in a program, one has a powerful means of detecting and preventing program errors; attempting to operate on a piece of data in a way that is inconsistent with its typing will result in a type error, rather than simply performing some undefined operations. Where these type errors occur is a language design choice that falls on a spectrum between dynamic typing and static typing. In dynamically-typed languages such as Python or JavaScript, judgments about whether an operation is well-typed are made during a program's execution. In constrast, statically-typed languages such as Haskell or OCaml determine whether operations are well-typed at compilation, with the expectation that these operations remain well-typed during execution.

Type safety, broadly, refers to the resiliency of a programming language to type errors over the course of executing a program in that language. In the context of statically-typed languages, type safety is achieved by ensuring that the program written in the language cannot enter an ill-typed (commonly referred to as *stuck*) state during execution. A stuck program is one that has finished evaluating to its final result, but has no way to continue execution as defined by the typing and evaluation semantics of the language. In a stuck state, the behavior of the program if one attempts to continue execution is undefined [3]. In practice, hitting stuck states during execution might result in segmentation faults or the silent corruption of the program's state.

Proving that a language's typing and evaluation semantics exclude the possibility of reaching a stuck state is also referred to as proving the *soundness* of the language's type system. One strategy for proving the soundness of a language is to prove that the the theorems of Progress and Preservation hold for the language. *Progress*, at a high-level, states that the well-typedness of a program at some point in execution implies that the program may continue to be evaluated. *Preservation* states that the execution semantics of the language do not change the typing of the program; the typing for the program and its sub-components at any point in execution is consistent with that determined prior to execution [4]. Soundness arises from these two theorems; if the language's execution semantics always have a defined behavior for a well-typed program and the well-typedness of a program is always preserved during the course of execution, then one knows that a well-typed program will always have a defined execution behavior. The exact formulation of these theorems for a language is dependent on how the evaluation and typing semantics are modeled by the verifier.

3. MOTIVATION

The primary motivation for this proof is verify that the typing guarantees provided by Haskell are actually well-founded. Haskell has found real-world usage due to the many practical benefits afforded by its strong, static type system. Haskell programmers are able to precisely specify the kinds of data that are input and output from functions, and will be warned at compilation of any errors due to a mismatch of an argument with its annotated input types. Haskell's type system aids in refactoring larger code bases; in the case where one decides to add a field to a record datatype, for instance, Haskell can report before compilation all of the points where the programmer must edit his or her code in light of this change. Haskell's type system also allows for extensions such as GADTs that allow for richer expression of a program's intended operation in the compile-time verified portions of the language. Using these features requires one to have faith that the compile-time guarantees given by GHC regarding a Haskell program will be carried forward in that program's runtime behavior. Proving that Haskell's type system is indeed type safe would allow programmers to be more confident that the compile-time guarantees made by GHC will hold at runtime, and thus be more confident in relying on the benefits provided by static type checking in software engineering projects.

Demonstrating the efficacy of Haskell's static type system could also help to increase the adoption of it and other statically-typed languages in critical systems, which could help to in turn prevent catastrophic failures of these systems at runtime. Two recent software flaws which could have potentially been prevented through the use of safe static languages is the Shellshock bug. Shellshock (also known as Bashdoor) was a bug discovered in the behavior of Bash when it spawned a new instance of Bash. The parent instance exports locally-defined functions in the form of a string which, when executed by the child instance, is a valid Bash command for declaring the function. The bug in Shellshock arose from the fact that the child instance would execute any inherited strings that were prefixed with a syntactically correct function definition. A malicious user could input as an environment variable a string that contained a valid function definition followed by an arbitrary sequence of Bash commands, causing these commands to be executed whenever the string was exported to a new shell instance. This error arises because function definitions are represented in strings when passed between Bash instances. This representation, while allowing one to encode a valid function definition as in the normal case for function export, also allows for malicious values to be represented [6]. This representation places the burden on the programmer to properly verify a string is not malicious or malformed; the existence of the Shellshock bug demonstrates that errors and oversights can happen in this situation, with disastrous consequences. This issue would be avoided in a safe language such as Haskell, as one would be able to pass function definitions between Bash instances using some datatype representation that can *only* represent function definitions. The type system could make it impossible to express a function definition that also executes some other malicious code, thus sidestepping the issue of this error.

This work is also motivated by the future extensions that can be made to the proofs. One area of direct extension is in proving that language extensions made to System FC in order to support new features in Haskell are type safe. Providing formal proofs of such features becomes increasingly relevant as the scope and complexity of System FC (and thus, the potential for errors in its formulation and verification) increases. Another area of direct extension for these proofs is in proving that the safety of various compiler optimizations, which risk changing the evaluation semantics of a given piece of code.

4. RELATED WORK

This work is built upon the idea of formal verification, wherein one generates a formal model of the system under study using a theorem-proving system such as Coq, and then proves that this model satisfies certain desired properties [2]. This methodology has been developed as an alternative to other program verification methods, such as model-checking, static analysis, or unit testing. This development was intended to sidestep the issue that it is either computationally infeasible or impossible to provide a guarantee of the system's correctness and safety using other verification methods.

Prior work has demonstrated that programming languages are targets for verification using formal methods. There exist full specifications and verifications for simple programming models, such as the simply-typed lambda calculus, as proven in [4] for example. In particular, it is desirable to verify the intermediate representation languages for compilers because the correctness of compilers is crucial in order to correctly execute and reason about programs written in that language [10].

Previous work has been done to formally verify the type safety of System F_ω, or System F with subtyping [7]. This formalization was done in Coq using De Bruijn indices. Additionally, some work has gone into formalizing the specification for System FC, including a basic Coq translation provided with the initial proposal of System FC [5]. Coercions themselves are also a very rich research topic, and a lot of previous work regarding coercions exist. Among other things, [1] and [9] provide a detailed concrete syntax of coercions as well as their inference rules. Though non-mechanized proofs of the type soundness of System FC exist [5], there has not been any substantial progress in a formal proof of the type soundness of System FC, or even System F with coercions.

5. SYSTEM MODEL

At a high-level, the system is composed of a formalization of the semantics of System FC, and a series of proofs of the properties of this formalization. The formalization itself is a representation of the System FC language using the language features of Coq. This is accomplished by defining inductive datatypes to serve as Coq correlates to the types (universally quantified types, function types, etc.) and terms (variables, type abstractions, etc.) introduced in System FC. With its abstract syntax defined, one can in theory construct a representation of any System FC program with its Coq formalization. Actually modeling the execution of this is accomplished by translating the execution semantics of System FC to Coq. This is done by creating in Coq a series of relations and fixpoints (the Coq terminology for a function, like those defined in Haskell or OCaml) that map terms to terms and types to terms according to the execution rules of System FC. With this formalization of the execution semantics, one is able to create a representation of any System FC program and then simulate its execution in the context of Coq. More importantly, the complete formalization of the operational semantics for System FC allows for proofs to be written in Coq about these operational semantics, which are needed to mechanically demonstrate the type soundness of System FC.

6. IMPLEMENTATION

The proof is implemented using the Coq Proof Assistant. First, the language semantics are formalized in the Coq language and then the key theorems are proven based on this formalization. The main theorems are the Progress, Preservation, and Soundness theorems.

6.1 Formalization

The formalization of System F in Coq follows directly from the definitions in [3]. The formalization of coercions comes from [1] and [9]. First, types, coercions, terms, and values are defined as inductive datatypes. Inductive datatypes are datatypes over which Coq can perform inductive proofs. This ability to perform inductive proofs is useful when proving the Progress and Preservation theorems for the formalization. The formalization of System FC syntax in Coq is shown in Figure 6 where `tm` represents a System FC term, `ty` represents a System FC type, and `cn` represents a System FC coercion. This formalization is modeled after the System FC term and context syntax in Figure 4 and System FC type and coercion sytax in Figure 12.

t	$::=$		**Terms**
	$\mid$	x	Variables
	$\mid$	$\lambda x : \tau . t$	Term Abstraction
	$\mid$	$t_1\ t_2$	Term Application
	$\mid$	$\Lambda \alpha . t$	Type Abstraction
	$\mid$	$t\ [\tau]$	Type Application
	$\mid$	$\lambda c : \varphi . t$	Coercion Abstraction
	$\mid$	$t\ \gamma$	Coercion Application
	$\mid$	$t \triangleright \gamma$	Safe Casting
u	$::=$		**Uncoerced Value**
	$\mid$	$\lambda x : \tau . t$	Term Abstraction
	$\mid$	$\Lambda \alpha . t$	Type Abstraction
	$\mid$	$\lambda c : \varphi . t$	Coercion Abstraction
v	$::=$		**Value**
	$\mid$	u	Uncoerced Value
	$\mid$	$u \triangleright \gamma$	Coerced Value
Γ	$::=$		**Contexts**
	$\mid$	$\varnothing$	Empty Context
	$\mid$	$\Gamma,\ x : \tau$	Term Variable Binding
	$\mid$	$\Gamma,\ \alpha$	Type Variable Binding
	$\mid$	$\Gamma,\ c : \varphi$	Coercion Variable Binding

Figure 4: System FC term and context syntax

τ	$::=$		**Types**
	$\mid$	α	Variables
	$\mid$	$\forall \alpha . \tau$	Polymorphic Types
	$\mid$	$(\sigma_1 \sim \sigma_2) \Rightarrow \tau$	Coercion Polymorphism
	$\mid$	$\tau_1 \rightarrow \tau_2$	Arrows
γ	$::=$		**Coercions**
	$\mid$	c	Variables
	$\mid$	$\langle \tau \rangle$	Reflexivity
	$\mid$	$\mathbf{sym}\gamma$	Symmetry
	$\mid$	$\gamma_1 \fatsemi \gamma_2$	Transitivity
	$\mid$	$\gamma_1 \rightarrow \gamma_2$	Arrows
	$\mid$	$\forall \alpha . \gamma$	Type Polymorphism
	$\mid$	$(\gamma_1 \sim \gamma_2) \Rightarrow \gamma$	Coercion Polymorphism
	$\mid$	$\mathbf{nth}^i \gamma$	Nth Argument Projection
	$\mid$	$\gamma_1\ [\gamma_2]$	Instantiation

Figure 5: System FC type and coercion syntax

```
(** *** Types *)
Inductive ty : Type :=
  | TVar    : nat -> ty
  | TArrow  : ty -> ty -> ty
  | TUniv   : ty -> ty
  | TCoerce : ty -> ty -> ty -> ty.

(** *** Coercions *)

Inductive cn : Type :=
  | CVar     : nat -> cn
  | CRefl    : ty -> cn
  | CSym     : cn -> cn
  | CTrans   : cn -> cn -> cn
  | CArrow   : cn -> cn -> cn
  | CTCoerce : cn -> cn -> cn -> cn
  | CNth     : nat -> cn -> cn
  | CTAbs    : cn -> cn
  | CTApp    : cn -> ty -> cn.

(** *** Terms *)
Inductive tm : Type :=
  | tvar    : nat -> tm
  | tapp    : tm -> tm -> tm
  | tabs    : ty -> tm -> tm
  | ttapp   : tm -> ty -> tm
  | ttabs   : tm -> tm
  | tcapp   : tm -> cn -> tm
  | tcabs   : ty -> ty -> tm -> tm
  | tcoerce : tm -> cn -> tm.

(** *** Values *)
Inductive uncoerced_value : tm -> Prop :=
  | uv_abs : forall T t,
      uncoerced_value (tabs T t)
  | uv_tabs : forall t,
      uncoerced_value (ttabs t)
  | uv_cabs : forall t T1 T2,
      uncoerced_value (tcabs T1 T2 t).

Inductive value : tm -> Prop :=
  | v_uncoerced : forall t,
      uncoerced_value t ->
      value t
  | v_coerced : forall t c,
      uncoerced_value t ->
      value (tcoerce t c).
```

Figure 6: System FC syntax defined in Coq

In this formalization of the language, variables are parameterized by a natural number instead of a string as one would normally expect. Generally, variables are named with strings, but two different variables could share a name, which causes issues when reasoning about a language using Coq. Conflicting variable names creates a problem for handling shadowing and substitution. For example, when given a function $(\lambda x : \mathsf{Int}.\ x + y)$ and attempting to substitute some externally defined variable x for all instances of y, problems arise. Performing a naïve substitution of $(\lambda x : \mathsf{Int}.\ x + y)[x/y]$ results in a function $\lambda x : \mathsf{Int}.\ x + x$. This changes how the function evaluates, so it does not preserve program semantics. Continuing without addressing this source of error in the specification of the operational semantics of the language would make proving the type system to be sound impossible.

There are several ways to get around this substitution issue. The way substitution is handled in the system described is through the use of De Bruijn indices [7]. This means that variables are encoded with natural number values. The value represents the number of binders between the variable's use and its binding site. Given this representation of variables, there cannot be naming conflicts. Consider the following example:

$$\Lambda T.\, \Lambda U.\, \lambda f : T \to U.\, \lambda x : T.\, f\ x$$

which has type:

$$\forall T.\, \forall U.\, (T \to U) \to T \to U$$

Using De Bruijn indices for both types and terms, the expression becomes:

$$\Lambda.\, \Lambda.\, \lambda : 1 \to 0.\, \lambda : 1.\, \overline{1}\,\overline{0}$$

which has type:

$$\forall.\, \forall.\, (1 \to 0) \to 1 \to 0$$

Note that the above uses bars to describe identifiers for terms. The 0 refers to the closest type variable binder and $\overline{0}$ refers to the closest term variable binder. The first expression with traditionally named variables is a simple pair of type abstractions containing a pair of term abstractions related to the types introduced in the outer type abstractions. The variables must be explicitly named at the binding site. This is noted by the $\Lambda T.$ or $\lambda x : T$. Using De Bruijn indices eliminates this need, as referring to a binder just requires determining the distance from the usage site to the binding site. The $\Lambda T.$ in the first example is replaced simply by $\Lambda.$ and the $\lambda x : T$ is transformed into $\lambda : 1$. This is more clear in Figure 6, where `tabs` and `ttabs` are not bound with associated identifiers as they would be with named variables. The 1 here replaces the T, as there is a single type variable binder between the usage of the type and the desired binder.

This representation of variable names using De Bruijn indices is noted in the formalization in Figure 6 in the variable cases in *ty*, *tm*, and *cn*. All of these definitions take in a natural number as the variable name. This formalization affects the proofs through the rest of the implementation, most notably introducing the need for index shifting. Take, for example, the substitution $[Y \mapsto X](\Lambda X.\, X \to Y)$ where X is to be substituted for Y in the type abstraction. Using De Bruijn indices, assume the outer X is known as 2 and the outer Y is known as 3 outside of the lambda. If this were the case, then the inner X would have to be 0 using De Bruijn indices, as it refers to the closest binding lambda to it. The inner Y would then refer to the same binder as the outer Y. Even though the outer Y is known as 3 outside of the type abstraction, it must be known as 4 inside of the abstraction, as an extra binder is introduced between the two uses of Y. Despite the confusion this may introduce, this makes the substitution $[3 \mapsto 2](\Lambda.\, 0 \to 4)$. This is where shifting is relevant. A naïve substitution would look for all instances of 3 and replace them with 2. This is not correct, though, as the search for uses of 3 passes a binding site for type variables as it enters the lambda. This means that what used to be 3 binders away from the binding site is now 4 away. Thusly, shifting the index of the substituted variables is necessary to preserve program semantics and adequately work toward a proof of the type safety of System FC.

In order to handle the several forms of substitutions that can occur in the language, substitution is defined as Coq Notation. Notation in Coq allows the user to refer to fixpoints (functions) and datatypes using operators instead of names. In this implementation, the notation for substitutions is `[x := s]t` meaning substitute the identifier `x` with `s` in `t`. There are a number of different types of substitutions that an occur in the language between all permutations of types, terms, and coercions, meaning that `s` and `t` are not always the same type. In order for the substitution notation to be be reusable for the different kinds of substitutions, there is a `Subst` typeclass that has instances for the combinations of types for `x`, `s`, and `t`. In addition to being defined using fixpoints, substitutions are defined inductively. Fixpoints provide a nice notation for substitutions while the inductive definition is useful in proofs. In order to reconcile this, several lemmata are proven that verify the equivalence between the fixpoint and inductive definitions for substitutions. This allows theorems to use the more readable notation, while the proofs of those theorems use the inductive definitions.

$\boxed{t_1 \longrightarrow t_2}$ **One-Step Reduction**

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$ Term Reduction

$$\frac{}{(\lambda x : \tau.t)\ t' \longrightarrow t[t'/x]}$$ Term Application

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [\tau] \longrightarrow t_1'\ [\tau]}$$ Type Reduction

$$\frac{}{(\Lambda\alpha.t)\ [\tau] \longrightarrow t[\tau/\alpha]}$$ Type Application

$$\frac{t_1 \longrightarrow t_1'}{t_1\ \gamma \longrightarrow t_1'\ \gamma}$$ Coercion Reduction

$$\frac{}{(\lambda c : \varphi.t)\ \gamma \longrightarrow t[\gamma/c]}$$ Coercion Application

$$\frac{t_1 \longrightarrow t_1'}{t_1 \rhd \gamma \longrightarrow t_1' \rhd \gamma}$$ Casting Reduction

$$\frac{}{(t \rhd \gamma_1) \rhd \gamma_2 \longrightarrow t \rhd \gamma_1 \mathbin{;} \gamma_2}$$ Coercion Transitivity

$$\frac{\varnothing \vdash v : \sigma_1 \to \sigma_2}{\begin{array}{c}(v \rhd \gamma)t \longrightarrow \\ v(t \rhd \mathbf{sym}(\mathbf{nth}^1\gamma)) \rhd \mathbf{nth}^2\gamma\end{array}}$$ Term Application Push

$$\frac{\varnothing \vdash v : \forall\alpha.\tau}{(v \rhd \gamma)\ [\sigma] \longrightarrow v\ [\sigma] \rhd \gamma\ [\sigma]}$$ Type Application Push

$$\frac{\begin{array}{c}\gamma' = \mathbf{nth}^1\gamma_0 \mathbin{;} \gamma \mathbin{;} \mathbf{sym}(\mathbf{nth}^2\gamma_0) \\ \varnothing \vdash v : (\sigma_1 \sim \sigma_2) \Rightarrow \tau\end{array}}{(v \rhd \gamma_0)\gamma \longrightarrow v\ \gamma' \rhd \mathbf{nth}^3\gamma_0}$$ Coercion Application Push

Figure 7: System FC evaluation rules

After formalizing syntax and substitutions for System FC, the evaluation semantics of System FC are then defined via the step relation. In order to ensure the safety of a type system, some formalization of how expressions in the language evaluate is needed to reason about the full execution of a program in the target language. The described formalization takes from the small-step language semantics in [4] to define how a single expression can take a single step toward its final value. The evaluation rules are detailed in Figure 7. This is defined inductively with the associated notation `t1 ==> t2` meaning that term `t1` steps to a term `t2`.

Variable binding contexts are implemented as an inductive datatype where `empty` represents the empty context and the data constructors `ext_var` and `ext_tvar` extend a context with a term or type variable binding respectively. Since variables are identified by de Bruijn indices, there is no need to specify any kind of identifier when binding a variable. By definition, a variable with de Bruijn index n is bound n levels up from the current scope. This means that the binding is the n^{th} deep type/term/coercion variable binding in the context `Gamma` [7].

$\boxed{\Gamma \vdash \gamma : \varphi}$ **Coercion Typing**

$$\frac{\vdash \Gamma \quad c : \varphi \in \Gamma}{\Gamma \vdash c : \varphi} \qquad \text{Coercion Variables}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash \tau}{\Gamma \vdash \langle\tau\rangle : \tau \sim \tau} \qquad \text{Reflexivity}$$

$$\frac{\Gamma \vdash \gamma : \sigma_1 \sim \sigma_2}{\Gamma \vdash \mathbf{sym}\gamma : \sigma_2 \sim \sigma_1} \qquad \text{Symmetry}$$

$$\frac{\begin{array}{c}\Gamma \vdash \gamma_1 : \sigma_1 \sim \sigma_2 \\ \Gamma \vdash \gamma_2 : \sigma_2 \sim \sigma_3\end{array}}{\Gamma \vdash \gamma_1 \mathbin{⨾} \gamma_2 : \sigma_1 \sim \sigma_3} \qquad \text{Transitivity}$$

$$\frac{\begin{array}{c}\Gamma \vdash \gamma_1 : \forall\alpha.\tau_1 \sim \forall\alpha.\tau_2 \\ \Gamma \vdash \gamma_2 : \sigma_1 \sim \sigma_2\end{array}}{\Gamma \vdash \gamma_1\ [\gamma_2] : \tau_1[\sigma_1/\alpha] \sim \tau_2[\sigma_2/\alpha]} \qquad \text{Instantiation}$$

Figure 8: Selected System FC coercion typing rules

With contexts defined, it is possible to define the final piece of the System FC formalization, typing. Typing is defined as an inductive proposition called `has_type`. A notation is also defined for the typing relation: `Gamma |- t \in T` means that the term `t` has type `T` under the context `Gamma`. Since the type of this expression in Coq is `Prop` (short for proposition), it is an expression that must be true if it can be proven in Coq. Given the formalization presented above, it is possible to prove that any well typed term `t` with type `T` indeed has type `T`. If `t` is not well-typed, then it is impossible to prove that it has any type. The typing rules for terms are displayed in Figure 9 and for coercions are displayed in Figure 8.

6.2 Progress

The next step after fully formalizing System FC is to prove the Progress theorem. The formal definition of Progress in Coq can be seen in Figure 10. More intuitively, the Progress theorem states that any term that is well-typed under the empty context is either a value or it can take a step. The proof of progress is by induction on the typing derivation `empty |- t \in T`. Recall that the `has_type` relation is defined inductively, therefore it is possible to proceed with an inductive proof over the relation.

The proof is laid out so as to have a separate case for every constructor in the inductive definition of `has_type`. The cases for the typing relations of values represent base cases in the inductive proof. In the other cases, Coq provides induction hypotheses that must be applied in order to complete the proof. Most of the cases can be automated by Coq; the goals are straightforward to prove. The most involved cases are term application, type application, and coercion application. These are more complicated because it is necessary to handle the sub-cases where the terms, types, or coercions in the application could be values or could step.

$\boxed{\Gamma \vdash t : \tau}$ **Term Typing**

$$\frac{\vdash \Gamma \quad x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{Variables}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . t : \tau_1 \to \tau_2} \qquad \text{Term Abstraction}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \tau_1 \to \tau_2 \\ \Gamma \vdash t_2 : \tau_1\end{array}}{\Gamma \vdash t_1\ t_2 : \tau_2} \qquad \text{Term Application}$$

$$\frac{\Gamma, \alpha \vdash t : \tau}{\Gamma \vdash \Lambda\alpha.t : \forall\alpha.\tau} \qquad \text{Type Abstraction}$$

$$\frac{\begin{array}{c}\vdash \Gamma \quad \Gamma \vdash \sigma \\ \Gamma \vdash t : \forall\alpha.\sigma\end{array}}{\Gamma \vdash t\ [\sigma] : \tau[\sigma/\alpha]} \qquad \text{Type Application}$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma, c : \varphi \vdash t : \tau}{\Gamma \vdash \lambda c : \varphi . t : \varphi \Rightarrow \tau} \qquad \text{Coercion Abstraction}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \varphi \Rightarrow \tau \\ \Gamma \vdash \gamma : \varphi\end{array}}{\Gamma \vdash t\ \gamma : \tau} \qquad \text{Coercion Application}$$

$$\frac{\begin{array}{c}\Gamma \vdash \gamma : \tau_1 \sim \tau_2 \\ \Gamma \vdash t : \tau_1\end{array}}{\Gamma \vdash t \triangleright \gamma : \tau_2} \qquad \text{Safe Casting}$$

Figure 9: System FC term typing rules

```
Theorem progress : forall t T,
      empty |- t \in T ->
      value t \/ exists t', t ==> t'.
```

Figure 10: The Progress Theorem written in Coq

6.3 Preservation

The proof of Preservation is significantly more involved than the proof of Progress. The formal definition of Progress in Coq can be seen in Figure 11. The Preservation theorem states that any term will preserve its type when taking a step. If a term has type `T` and then takes a step, the resulting term will also have type `T`.

In order to prove Preservation, many lemmata must first be defined and proven. Among these, several weakening and strengthening lemmata help one reason about the well-formedness of a context, type, or coercion with a variable ex-

```
Theorem preservation : forall Gamma t t' T,
    Gamma |- t \in T ->
    t ==> t'          ->
    Gamma |- t' \in T.
```

Figure 11: The Preservation Theorem written in Coq

$\boxed{\vdash \Gamma}$ **Well Formed Contexts**

$$\frac{}{\vdash \varnothing} \quad \text{Empty Context}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau} \quad \text{Term Variable Binding}$$

$$\frac{\vdash \Gamma}{\vdash \Gamma, \alpha} \quad \text{Type Variable Bindings}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash \sigma_1 \quad \Gamma \vdash \sigma_2}{\vdash \Gamma, c : (\sigma_1 \sim \sigma_2)} \quad \text{Coercion Variable Bindings}$$

$\boxed{\Gamma \vdash \tau}$ **Well Formed Types**

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \quad \text{Type Variables}$$

$$\frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \quad \text{Arrows}$$

$$\frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha . \tau} \quad \text{Polymorphic Types}$$

$$\frac{\Gamma \vdash \sigma_1 \quad \Gamma \vdash \sigma_2 \quad \Gamma \vdash \tau}{\Gamma \vdash (\sigma_1 \sim \sigma_2) \Rightarrow \tau} \quad \text{Coercion Polymorphism}$$

Figure 12: System FC well-formedness rules for contexts and types

tended or removed from the context. These well-formedness rules are included in Figure 12. Also included in the system are several substitution lemmata which are used to prove that substituting a type, term, or coercion in an expression preserves the type of the expression. Using these lemmata, the Preservation theorem is also proven by induction on the typing derivation `Gamma |- t \in T`. Much like the Progress theorem, the majority of the work done in the proof takes place in cases for term, type, and coercion application.

6.4 Soundness

The final theorem in proving the type safety of System FC is the Soundness Theorem. The formal definition of Soundness in Coq can be seen in Figure 13. This theorem states that at any point in a well-typed term's execution, after any number of discrete steps, the term will not be in a stuck state, where a stuck state is any state where the term is not a value but also cannot take a step. More explicitly, this theorem says that if the type checker considers a term well-typed, then it will never get stuck in some inconsistent state, which is one standard definition of type safety. The proof of the Soundness theorem is done by induction on the number of discrete steps taken and a straightforward application of the Progress and Preservation theorems.

```
Theorem soundness : forall t t' T,
    empty |- t \in T ->
    t ==>* t'         ->
    ~(stuck t').
```

Figure 13: The Soundness Theorem written in Coq

7. RESULTS

The system as it stands is a full mechanical proof of the type safety of System F with coercions. The type safety of all features of System FC have previously been proven by hand [5], though it has never been done mechanically using a proof assistant. The system described is the first mechanical proof of this particular subset of System FC, namely System F with coercions. Other mechanical proofs of the type safety of System F exist, though none contain a formalization of coercions as described in [1] and [9]. Given this, it is the first result that verifies that System F with coercions is type safe.

8. FUTURE WORK

As the formalization stands, it does not contain all features commonly thought to be part of System FC. In order to reach feature parity with the full System FC, it would need to be extended to include data types and type families. These two additions would create the full formalization of System FC. It would then be necessary to adjust the proofs accordingly so that the Coq interpreter would accept them. A proof of the type safety of this more complete version of System FC would be a big step toward a proof of the type safety of Haskell. Recall that System FC is a specification for GHC Core, the actual implementation used in GHC, so a correctness proof of the translation from System FC to GHC Core would be necessary to have a proof of the type safety of System FC as used in the Haskell compilation process.

The mechanical proof as it stands is very extensible. This allows for a large number of potential extensions and use cases. Modifying a mechanical proof can be done by adding to an already proper formalization and then walking through the proof with the help of the Coq interpreter to fix anything broken by the change to the formalization. Given that most of the proofs in the system are done by induction on some inductive definition in the formalization, adding to the formalization would in most cases only add new cases, and much of the proof structure and the correctness of many of the proofs would be preserved.

With a full formalization and proof of System FC, possible extensions include verifying the safety of various GHC language extensions. GHC includes many features, but programmers are given even more options via extensions to the language. These can be included by annotating the program or adding command-line flags. Many commonly used language extensions add to Haskell's type system, so being able to prove the type safety of an extension would allow for an added level of guarantees to the potentially unsafe extension.

A mechanical proof of the type safety of GHC Core could also be used to check properties of compiler optimizations.

At the GHC Core stage of GHC's compilation process, GHC does many of its optimization passes. Using the formalization and proof of type safety, one could extend the formalization to include a set of compiler optimizations. These optimizations could then be proven to be type safe to ensure the semantics of a program cannot be changed through an invalid optimization done at the GHC Core level.

9. ETHICS

As discussed above, one of the primary areas of future work with this formalization is using it as the foundation for future verification of optimizations and language extensions. These use cases raise the concern that the presented formalizations and thus presented proofs are incorrect. In the case where this work were to falsely suggest that System FC is type-safe, one would be left with the concern that numerous language extensions, compiler systems, and software systems were constructed on the false premise that their underlying type system was not inherently flawed. The real-world impact of these flaws is dependent on how likely it is for runtime systems to reach an inconsistent state in the flawed type system and how critical the systems that demonstrate these flaws are.

Another potential issue arises in the case where this work were to falsely suggest that System FC is not type-safe. This would present a major breach of the trust people place in programming languages such as Haskell that are reliant on System FC, potentially leading to the adoption of less safe languages for critical systems. As discussed earlier, the adoption of unsafe languages for critical systems can have disastrous consequences, allowing whole classes of program errors to be uncaught and potentially exploited by malicious parties.

10. CONCLUSION

Formal verification, as a tool, allows one to verify and prove properties of systems with a rigor and explicitness much greater than that afforded by other techniques. As such, it is worthwhile to use formal verification to prove the core assumptions behind the software systems that we use. We have demonstrated that a significant portion of System FC, the theoretical basis of the Haskell programming language, is sound; while we cannot yet rule out all logical errors that can be introduced by the programmer, we can now rest assured that a compiling Haskell program should not fall into a state of undefined behavior during its execution. Along with demonstrating the correctness of Haskell today, this proof paves the way for the formal verification of the more advanced and more expressive type systems of tomorrow.

11. REFERENCES

[1] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 189–202, New York, NY, USA, 2014. ACM.

[2] Patrick Cousot and Radhia Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In Javier Esparza, Bernd Spanfelner, and Orna Grumberg, editors, *Logics and Languages for Reliability and Security*, volume 25 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 1–29. IOS Press, 2010.

[3] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[4] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2014.

[5] Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In François Pottier and George C. Necula, editors, *TLDI*, pages 53–66. ACM, 2007.

[6] Shellshock bug has security experts in a panic and hackers searching for exploits. *Network Security*, 2014(10):1 – 2, 2014.

[7] Jérôme Vouillon. A solution to the poplmark challenge based on de bruijn indices. *J. Autom. Reasoning*, 49(3):327–362, 2012.

[8] Dimitrios Vytiniotis and Simon L. Peyton Jones. Evidence normalization in system FC (invited talk). In *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, pages 20–38, 2013.

[9] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon L. Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 53–66, 2012.

[10] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 427–440, New York, NY, USA, 2012. ACM.