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's compiler, the Glasgow Haskell Compiler (GHC), generates code in GHC Core. The Coq proof assistant will be used to verify a formalized version of System FC, the basis for GHC Core. A translation from the formal language to GHC Core, the concrete implementation of System FC that is used in GHC, will then be proven. The goals of verification are to prove that the evaluation semantics of System FC are sound.

There are two main benefits to this project. First, the verification would provide assurance regarding the safety and accuracy of GHC. Second, and perhaps more importantly, it will provide foundation to verify other properties of GHC such as compiler optimizations.

1. INTRODUCTION

Haskell has one of the strongest type systems of any mainstream programming language with features such as type families, typeclasses, and generalized algebraic datatypes (datatypes that have non-standard constructors). When writing Haskell, there are many guarantees of correctness encoded in the type system. Ensuring that the type safety of features like these is preserved in System FC would be greatly beneficial as it would act as a proof of correctness for the language. A core language is a translation from a surface language (a language that a programmer can better reason about) to a language that a compiler can better reason about. Whereas types are important in programming languages to prevent a user from making errors, types are important in core languages to prevent the compiler from making errors. This type safety will be ensured by proving the progress, preservation, and soundness theorems using the formalized definition of System FC.

There is no formal proof that GHC Core is semantically correct. The plan is to provide such a proof. The formally verified version of System FC will allow for future work in the formal verification of GHC. Once the core language is verified, it becomes possible to verify that transformations of this core language preserve the typing and progression semantics of the original language [9]. One particularly relevant class of transformations on the core language is the set of compiler optimizations performed by GHC. While these transformations can improve the runtime of one's code, they can also introduce bugs due to the semantics of the original code not being preserved under the optimization. With the semantics of the source language formalized and verified, it becomes easy to extend the formalizations to encompass the changes made under these optimizations, and thus verify their correctness [10]. Another major use would be the ability to extend the proof, in order to verify extensions to System FC. Such extensions to System FC are used to add new features to the surface language.

To accomplish this, GHC Core is verified in a series of steps. First, System F is verified. Next, coercions are added. Following this, the system is enhanced with the addition of type families, and then datatypes.

2. BACKGROUND

Progress, preservation, and soundness are the most basic indications of safety for any type system [4]. To understand these, it is important to define how the operational semantics of a language are formalized. Types are descriptors for data and terms are well-typed expressions under a given context, or set of typing assumptions. When specifying the operational semantics of a programming language, one defines a step relation that relates a term of a particular type in the language to another term of the same type. Terms make steps in order to reach a simplified form, see Figure 1 for an example. The evaluation of any expression in the language can be modeled in a series of discrete "steps", where in each step one takes a term in the expression and replaces it with the resulting term per the step relation. In a well-typed program, this process continues until the expression evaluates to a value, a unique representation of a value for a particular type (e.g. '1' for the type of integers).

$$(\lambda x.\ \lambda y.\ x + y)\ 3 \rightarrow \lambda y.\ 3 + y$$

Figure 1: Concrete example of a the step relation for a function application, known formally as Beta Reduction.

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

The progress, preservation, and soundness theorems can now be defined. Progress states that a well-typed term is either a value, or can take a step per the step relation. Preservation indicates that if a well-typed term takes a step, the resulting term will still be well-typed. If something cannot take a step and is not a value, the program is said to be stuck. Combining these definitions, soundness states that a well typed program will never become stuck [4]. This stuckness will cause errors including but not limited to a segmentation fault.

Theorem (Progress) *For all terms t, types T, and contexts Γ, if t has type T under context Γ then either t is a value, or there exists a term t' such that t steps to t'*

Theorem (Preservation) *For all terms t and t', types T, and contexts Γ, if t has type T under the context Γ and t steps to t' then t' has type T under the context Γ.*

Theorem (Soundness) *Any well-typed program will never reach a stuck state during evaluation.*

Together, these properties guarantee that a well-typed program in the specified language will always be able to continue evaluation until it reaches a value; the result of the program ending up in a stuck state will only result from the program logic leading to an infinite loop, and not from the language falling into some invalid state. After formalizing the operational semantics for System FC, it will be demonstrated that preservation, progress, and soundness hold for the specification.

System FC is built on top of the simpler language System F. System F, also known as the polymorphic lambda calculus, is an extension of the simply-typed lambda calculus to include the abstraction and application of type terms [4]. 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. System F is formalized in Coq and then additional features are added in order to transform System F into a full formalization of System FC. These features include type coercion, type families, and datatypes. In particular, 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 [7]. For an example of coercions, see Figure 4. 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]. System F, with these added features, becomes system FC. A translation to System FC is then proven which shows that the core language of GHC has indeed been verified. The syntax, evaluation semantics and type relations for System F are provided in Figure 2.

Syntax

$t ::=$		**Terms:**
	x	*variable*
	$\lambda x : T.t$	*abstraction*
	$t\ t$	*application*
	$\lambda X.t$	*type abstraction*
	$t\ [T]$	*type application*
$v ::=$		**Values:**
	$\lambda x : T.t$	*abstraction value*
	$\lambda X.t$	*type abstraction value*
$T ::=$		**Types:**
	X	*type variable*
	$T \to T$	*type of functions*
	$\forall X.T$	*universal type*
$\Gamma ::=$		**Contexts:**
	$\varnothing$	*empty context*
	$\Gamma, x : T$	*term variable binding*
	Γ, X	*type variable binding*

Evaluation

$$\frac{t_1 \to t_1}{t_1'\ t_2 \to t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda x : T_{11}.t_{12})\ v_2 \to [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

$$\frac{t_1 \to t_1'}{t_1\ [T_2] \to t_1'\ [T_2]} \qquad \text{(E-TApp)}$$

$$(\lambda X.t_{12})\ [T_2] \to [X \mapsto T_2]t_{12} \qquad \text{(E-TAppTAbs)}$$

Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2\ :\ T_1 \to T_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-App)}$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \qquad \text{(T-TApp)}$$

Figure 2: Specification of System F as defined in Types and Programming Languages[4]

3. 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 [5] 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].

Further, some work has gone into formalizing the specification for System FC, including a basic Coq translation provided with the initial proposal of System FC [6]. Progress for System F has a non-mechanized proof [3]. To our knowledge, however, there has not been any substantial progress in a formal proof of the soundness of System FC.

4. PROJECT PROPOSAL

This section describes the proposed approach. It begins by detailing the set of steps we plan to follow in order to successfully verify System FC. Next, it describes the technical challenges that are involved in the project and finally, it explains the evaluation criteria.

4.1 Anticipated Approach

To begin, the semantics of System F as defined in [5] are formalized in Coq. Progress and preservation are then proven to hold for this formalization. Once the proofs of these theorems have been generated and mechanically verified, the formalization is extended to include the remaining features of System FC that are absent from System F. At each step the proofs are be adjusted to account for the features that have been added. In this way, the proofs are incrementally built until there is a complete proof for a formalized version of System FC. This approach is outlined in Figure 3. The first feature that is added to the formalization of System F is type coercions without datatypes. Coercions 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 in Figure 4.

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

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

Figure 4a: Haskell code where coercion is needed

```
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 4b: Translation of Haskell code in Figure 4a to System FC. The necessity of is demonstrated here.

This is a basic example of datatypes in Haskell and a trivial use of them. `G` is a parameterized datatype with kind $* \rightarrow *$. A *kind* in Haskell represents the type of a type constructor. In Haskell, the types `Int` and `Bool` both have kind `*`. `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 `*`, 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`.

Note here that datatypes are used to demonstrate the power of coercions. However, at this point in the implementation, there will not be datatypes. Datatypes are much more complicated to formalize than are coercions and type families, and it is easier to formalize datatypes after formalizing coercions and type families.

After adding type coercions without data types, and verifying correctness using the progress and preservation theorems, the type family feature is added. At the most basic level, type families are just functions at the type-level. Below is a basic example of a type family.

```
data Ty = Tint
       | TBool

type family I (t : Ty) :: *
type instance I TInt  = Int
type instance I TBool = Bool
```

In this example [8], the type `Ty` is defined to be either a `TInt` or `TBool`. Then, the type family `I` is defined to act on something of type `Ty`, either a `TInt` or `TBool`. This returns something of kind `*`. So, this type family `I` can map the data constructor `TInt` to `Int` and `TBool` to `Bool`.

Next, datatypes are added to the formalization of System FC. Datatypes are another very powerful construct in Haskell that allow programmers to define their own algebraic types. Types defined in such ways can be used to represent many constructs. For example, datatypes can be used to define generic lists as follows:

```
data List a = Nil
            | Cons a (List a)
```

Here, a list is parameterized by the generic type variable `a` and is constructed as either `Nil`, the empty list, or an element of type `a` consed on to a list of type `a`. This construct is extremely simple, but also very powerful. It is another very important addition to System FC.

The language resulting from adding the features described above to System F is a formalization of the language System FC. A translation to the core language that is actually implemented in GHC is then be proven. This verified translation then proves that the formalized language is indeed equivalent to System FC and therefore the proofs on the formalized language hold on the actual language.

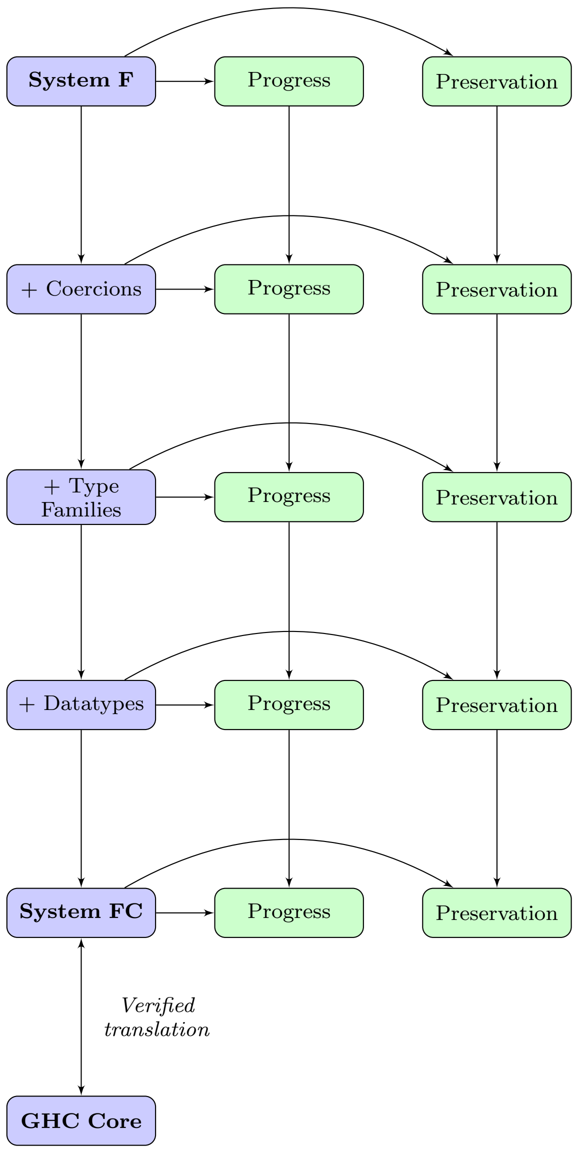

Figure 3: Block diagram depicting the anticipated approach. The arrows on the left side represent work required to complete each given step.

4.2 Technical Challenges

One of the primary challenges in completing this project is deciding on an order to iteratively add features in order to sensibly formalize System FC and GHC Core. Currently, the plan is to add coerciosn first, followed by type families, followed by datatypes. However, there may be unforeseen dependencies that make this ordering impractical. For example, it may be impossible or infeasible to add coercions before adding datatypes. Further, GHC Core would prove difficult to formalize due to the fact that the operational and typing semantics of this language are geared toward making it easy to perform compiler optimizations on the language, rather than making it easy to analyze its semantics. In order to obviate this, the semantics of System FC will be formalized and then a direct translation between these formalized semantics and the semantics of GHC Core will be proven.

One specific challenge may arise regarding the proofs of progress, preservation, and soundness in some instances after coercions, datatypes, and type families have been added. Recall Figure 4. Here, a coercion could be added in the scrutinee of the case statement of `f`, where the scrutinee of a case statement represents that which is being broken down into cases. With this addition, progress, preservation, and soundness may be very difficult to prove.

Additionally, since the formalization of System FC has never been performed before, it is difficult to estimate how long it will take to formalize different parts of System FC. In fact, axioms may be required in order to bypass rather large or unwieldy proofs that are obvious, but necessary to achieve the primary goals. Further, progress, preservation, and soundness may be, in fact, impossible to prove. This would be a result of the inconsistency in the specifications of System FC, which would be a major unexpected result.

4.3 Evaluation Criteria

This formalization for System FC has never been done. Upon completion, assurance of the formal correctness of this language will be provided, which is essential to the compilation of Haskell. This would also form a building block for verifications of other features of System FC. The primary criterion for evaluation would be how much of the specification of System FC is formalized; given that there exists no precedent for formalizing System FC, the completion of the full specification is not assured. The formalization of the base specification for System FC can be done in several parts, meaning the progress of this project can be judged based on what sections of the language specification are completed. Completing this, the project could be arbitrarily extended to formalize proposed extensions to System FC; evaluation of this could again be based on how many of these extensions are completed. Additionally, another criterion for evaluation would be how many axioms are used in the formalization. The fewer axioms assumed, the stronger the result becomes.

5. RESEARCH TIMELINE

- ALREADY COMPLETED: Understand the formal definition of System F.
- PRIOR-TO THANKSGIVING : Formalize System F in Coq.
- PRIOR-TO CHRISTMAS : Prove the substitution lemma holds in System F.
- COMPLETION TASKS : Complete verification of System F with coercions and type families.
- IF THERE'S TIME : Complete formalization of System FC, verify formal language, prove translation between formal language and System FC as implemented in GHC. Also verify other GHC features, such as optimizations and extensions.

6. 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] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

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

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

[6] 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.

[7] 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.

[8] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 275–286. ACM, 2013.

[9] J. Zhao. *Formalizing the SSA-based Compiler for Verified Advanced Program Transformations*. University of Pennsylvania, 2013.

[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.