CIS400/401 Project Proposal Specification [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

We plan to verify a formalized version of System FC, the core language of the Glascow Haskell Compiler (GHC), using the Coq proof assistant. We will then prove a translation from our formal language to GHC Core, the concrete implementation of System FC that is used in GHC. The goals of verification are to prove that the evaluation semantics of System FC are type safe.

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 optimization.

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. When writing Haskell, there are many guarantees of correctness encoded in the type system. We wish to ensure that the type safety of features like these is preserved in System FC, the GHC core language. We will do this by proving the progress, preservation, and soundness theorems using our formalized definition of System FC.

The formally verified version of System FC will pave the way 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. 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.

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

2. BACKGROUND

Progress, preservation, and soundness are the most basic indications of safety for any type system [1]. To understand these, we must first note how the operational semantics of a language are formalized. When specifying the operational semantics of a programming language, one defines a step relation that relates a term of a particular type in that language to another term. One can then model the evaluation of any expression in the language 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, terminating program, this process continues until the expression evaluates to a canonical form, a unique representation of a value for a particular type (e.g. '1' for the type of integers). A well-typed program can also fail to terminate by entering into an infinite loop, and an ill-typed language can fail to terminate by getting stuck, which happens when the step relation provides no simplification for the current term.

Knowing this, we can now define the progress, preservation, and soundness theorems. Progress states that a well-typed term is either a canonical form, 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 [1].

Theorem (Progress) *For all terms t, types T, and contexts Γ, if t has type T under context Γ then either t is a canonical form, 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) *A term t is not stuck if it is well types and it can take a step or it is a canonical form.*

Together, these properties guarantee that a well-typed program in the specified language will always be able to continue evaluation until it reaches a canonical form; the

failure of the program to terminate 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, we plan on demonstrating that preservation, progress, and soundness hold for the specification, which in turn will allow us to prove that the formalization is sound. We can then show that the program is sound if we can show using preservation, progress, and soundness that a well-typed term will never reach a stuck state, thus ensuring that the program remains in a valid state throughout its execution.

System FC is built on top of the simpler language called 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. 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. We will first formally verify System F in Coq and then we will add the additional features needed 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 [**?**]. Once we have added these features to System F, we will have a formalized language that is equivalent to System FC. We will then be able to prove a translation to System FC which will show that we have indeed verified the core language of GHC. The syntax, evaluation semantics and type relations for System F are provided below.

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

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

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

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

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

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

Typing

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

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

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

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

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

3. RELATED WORK

4. PROJECT PROPOSAL

4.1 Anticipated Approach

To begin, we plan to create a Coq formalization of the semantics of System F as defined in Types and Programming Languages. We will then prove that progress and preservation hold for this formalization. Once we have generated verified proofs of these theorems, we plan to extend our formalization to include the remaining features of System FC that are absent from System F. At each step we will adjust our verification to account for the features that we have added. In this way, we will incrementally build the proofs until we have arrived as a complete proof for a formalized version of System FC.

The first feature that we plan to add to our 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 for type-level abstraction and polymorphism, by allowing a conversion from one type to another. A basic example of the usefulness of coercions is below.

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

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

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. Specifically, in System FC, this same segment of code would look like this:

```
G :: * -> *
G1 :: forall (a :: *). a ~ Int -> G a
G2 :: forall (a :: *). a ~ Bool -> G a

f :: forall (a :: *). G a -> a
f = \(a :: *). \(x :: G a)
  case x of
    G1 c -> 5 ▷ sym c
    G2 c -> True ▷ sym c
```

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 our implementation, we will not have added datatypes. Datatypes are much more complicated to formalize than are coercions and type families, and it will be easier to formalize datatypes after formalizing them.

After adding type coercions without data types, and verifying correctness using the progress and preservation theorems, we plan to add the type family feature. 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, courtesy of [2], 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, we plan to add datatypes to our formalization of System FC. Datatypes are another very powerful construct in Haskell that allow programmers to define their own algebraic data types. Types defined in such ways can be used to represent virtually anything. For example, we can use datatypes 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.

Once we have added the features described above to System F, we will have a formalization of the language System FC. We will then prove a translation to the core language that is actually implemented in GHC. This verified translation will prove that our formalized language is indeed equivalent to System FC and therefore our proofs on the formalized language will hold on the actual language.

4.2 Technical Challenges

One of the primary challenges in completing this project is dividing the specifications of System FC and GHC Core in such a way that they can be sensibly formalized. GHC Core, in particular, 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 making it easy than analyze its semantics. We plan to obviate this by formalizing the semantics of System FC and then proving a direct translation between these formalized semantics and the semantics of GHC Core.

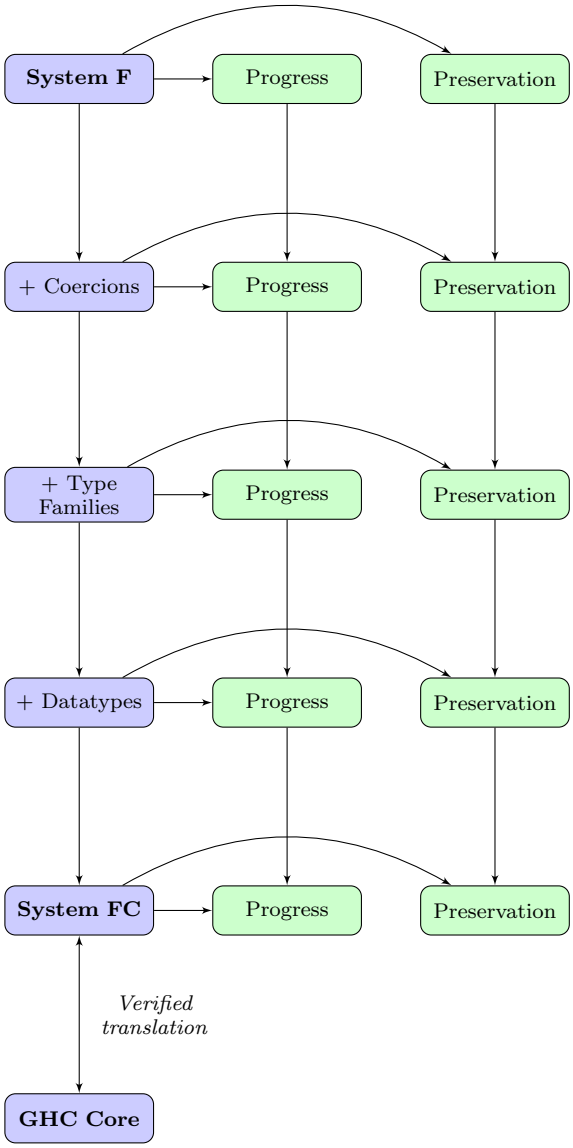
One specific challenge we may face regards proving progress, preservation, and soundness, in some instances after we have added coercions, datatypes, and type families. For example, recall the segment of code given in the previous section explaining coercions. Here, a coercion could be added in the scrutiny of the case statement of `f`, where the scrutiny 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. Further, we may find that progress, preservation, and soundness is 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, we will provide assurance of the formal correctness of this language, which is essential to the compilation of Haskell. We would also provide a building block for verifications of other features of System FC. Our primary criterion for evaluation would be how much of the specification of System FC we are able to formalize; 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 we can judge our progress based on what sections of the language specification are completed. Completing this, our project could be arbitrarily extended to formalize proposed extensions to System FC; evaluation of this could again be based on how much we complete of these extensions.

5. BLOCK DIAGRAM



6. RESEARCH TIMELINE

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

7. REFERENCES

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

[2] 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.