Practical Functional Programming with System F_C and its Extensions

APOORV INGLE, University of Iowa, USA

Programming languages are a primary interface between humans and computers. This makes it crucial to understand their principles of design and implementation. Programming languages need to encode unambiguous yet expressive instructions for computers. The features of a typed programming language strive to provide machine verified guarantees about the programs by following the "correct by construction" philosophy. In this monograph, we study the practical (type system) and meta-theoretical (formalization) aspects of a modern, statically typed, declarative, and functional programming language. We focus on how one single construct, explicit type equality, gives a unified account of several seemingly unrelated language features. As a consequence, we obtain a typed intermediate language that is an easy to reason about and implement. We also study important extensions of this language and identify notable open research problems.

1 Introduction

Constructive mathematics and computer programming have an elegant correspondence: they both involve solving a problem by identifying and then exploiting the right level of abstraction. The difference is in the direction. Mathematicians work from top to bottom; they first build the abstractions and then enrich them to represent real world problems. Programmers turn the process on its head. They first write a software program that solves a very specific task and then build the right level of abstraction by means of iterative code refactoring. This phenomenon is a well established observation in computer science: Curry-Howard or the Brouwer–Heyting–Kolmogorov correspondence [Han, 2023, Wadler, 2015] and it forms the basis for the field of type theory [Barendregt et al., 2013, Nordström et al., 1990, Univalent Foundations Program, 2013].

A programming language compiler, like any other software, evolves by means of extensions. A new language feature is implemented in two steps. First, the abstract syntax tree (AST)—the data structure that encodes the surface syntax of the language—is extended to represent the new feature. Second, the operations performed on the AST are extended to cater for this new feature. These operations transform the AST into an intermediate language internal to the compiler [Siek, 2023]. The intermediate language is more suitable representation of the surface level AST to perform code analysis and runtime optimizations [Aho et al., 2015]. If there is a way of encoding the new language feature by using existing language features, the second step can be skipped. However, not all language features can be encoded into existing language features. In such cases, the compiler writer, needs to enrich the intermediate language to support the new language feature. Extensions to the intermediate language, however, do not scale. Multiple extensions to the intermediate language makes it difficult to maintain, and reason about its correctness. The intermediate language requires a reformulation, which amounts to finding the right abstraction that can encode the different high level language features into a simplified intermediate language.

In a compiler for statically typed programming languages, the typechecker, is the gate keeper. Its purpose is to ensure that it does not let *bad* programs pass, and flag the appropriate offending parts of programs to the programmer. This gate keeping is desirable; it aids the programmer, by guaranteeing that a class of program errors, such as runtime null pointer exceptions due to incorrect function arguments, can never occur. They can then concentrate on the other, more

Author's Contact Information: Apoorv Ingle, University of Iowa, Department of Computer Science, Iowa City, Iowa, USA.

important aspects of the program, which cannot be verified by the typechecker. On the other hand, the typechecker needs to allow the *good* programs—the ones that are definitely bug free—to pass without the programmer's intense struggle. For this reason, the type structure needs to be sufficiently rich to express programmer's complex ideas and invariants. The examples below illustrate the invariants that the programmers may like to capture within the type structure:

(1) In an AST where the compiler writer wants to enforce the sub-term to always be of a fixed typed. This avoids the need to explicitly check them in the code. More concretely, the AST representing the control structures, such as **`if`** or **`while`** loop, is well formed only when the sub-term representing the condition is of type `Boolean`.
(2) In a multi-stage compiler, the AST is gradually enriched with more information after each analysis stage. The type structure can enforce that the information is accessed only after it is populated, or it raises a type error [Peyton Jones and Najd, 2017].

The thesis of this monograph is: *using explicit type equality, a novel construct, in the intermediate language is an appropriate abstraction to encode semantic invariants without compromising safety or efficiency.* Type equalities denote explicit proofs in the system which assert when two types can be considered synonymous. The monograph is arranged in four parts:

Part I serves as a reminder of the essential features of a statically typed functional programming language. This gives the *programmers' view* of the language motivated using real world examples.

Part II formalizes the syntax and operational semantics of System $\mathbf{F}_C$ which in essence is System **F** extended with typed equality. This part gives the *compiler writers' view* of the language. To this effect, we show how the language features described in Part I can be trivially encoded within System $\mathbf{F}_C$.

Part III describes two important extensions of vanilla System $\mathbf{F}_C$ which make it even more expressive: (1) System $\mathbf{F}_C^R$ makes type equality finer grained while providing stronger type soundness guarantees and (2) System $\mathbf{F}_C^\kappa$ squashes the distinction between types and kinds making the type (and kind) level computation even more expressive. Both of these extensions are formalized along with their correctness argument sketch.

Part IV describes notable open problems in the area. It is followed by important work adjacent to the principal topic of this monograph, and concluding remarks.

Part I: The Landscape

2 Program Behavior

2.1 Functions

There can be no functional programming language without a support for first class functions. Functions capture the essence of the program execution by encoding the logic of transforming data. An example of a function declaration in Haskell is that of an identity function, `idI` over integers and `idC` over characters shown below.

```
idI :: Int → Int
idI i = i
```

```
idC :: Char → Char
idC i = i
```

The type signature `idI :: Int → Int` specifies the input-output behavior of the function: `idI` accepts an argument of type `Int` and returns a value of type `Int`. The definition `idI i = i` specifies how the function satisfies the typing specification: it takes an argument `i` and returns it unmodified. Functions declared in a functional programming language resemble mathematical functions in two ways: first, they are referentially transparent, i.e. invoking a function with the same arguments will always return the same result: the expression, `idI 3`, will always evaluate to `3`. Second, they are declarative, in a sense they abstract away the implementation and execution details of how the function executes on the actual underlying hardware; there is no mention of allocating and freeing program memory or even an explicit return.

2.2 Parametric Polymorphism

Re-usablity of programs improves the programming experience. For example, an identity function over `Char`, `idC` has the exact same definition as `idI` over `Int`. The functions that work on base types, such as `Int`, are called monomorphic functions. To enhance re-usability of programs, it is necessary to abstract over types.

```
id :: t → t
id a = a
```

```
(∘) :: (a → b) → (b → c) → (a → c)
f ∘ g  = λ x → f (g x)
```

The function `id`, shown above, abstracts over all types by an implicit universal quantification over the type variable `t`. Such functions are called *parametrically polymorphic* functions [Strachey, 2000]. The compiler can deduce which instance of the polymorphic function is required when an argument of a concrete type is passed as an argument to a polymorphic function. The language is said to support functions as first class citizens when they can be used as arguments to other functions, new functions can be declared locally and also be able to compose without saturating them. For example, in the function declaration, which composes any two suitable functions, (∘), illustrates all these three properties. The arguments `f` and `g` are functions used as arguments, it declares an local anonymous function which binds the parameter `x`, `λ x → f (g x)`. ML [Milner, 1978, Milner et al., 1975] was the pioneer among the functional programming languages in introducing a declarative style implicit parametric polymorphism.

2.3 Adhoc Polymorphism

Consider the following two terms, `addI` and `addF`:

```
addI :: Int = 1 + 2
```

```
addF :: Float = 1.1 + 2.3
```

In the definition of `addI`, the operator (+) is applied to two integers, but in the definition of `addF`, the operator (+) is applied to two floating point numbers (`Float`). Although the programmer uses the same symbol, the meanings of the two terms are distinct: `addI` adds two `Int`s and returns an `Int` while the `addF` adds two `Float`s and returns a `Float`. The low level compiler generated code for each of them would also differ as they would make a call to two different built-in subroutines. This name punning technique, which is dependent on the type of arguments, or the context of where the function appears, is called *adhoc polymorphism* [Strachey, 2000], while the functions themselves are referred to as adhoc polymorphic functions. Implicit operator overloading is a mechanism to support adhoc polymorphism. The compiler resolves the overloaded operator (+) to the actual operator (`int_plus` or `float_plus`) during the typechecking phase.

```
class Num a where
  (≤) :: a → a → Bool
  (==) :: a → a → Bool
  (+) :: a → a → a
```

```
instance Num Int where
  (≤) = int_le
  (==) = int_eq
  (+) = int_plus
```

```
instance Num Float where
  (≤) = float_le
  (==) = float_eq
  (+) = float_plus
```

Fig. 1. `Num` typeclass and instances

To make implicit overloading usable in functional programming languages, the type checker should be able to rule out terms such `id + id`—summing 2 parametrically polymorphic functions is meaningless. Functions like `(+)` act only on some specific class of constrained or qualified types [Jones, 1994]: The addition function (+), more precisely, has the type `Num t ⇒ t → t → t`. This means that the function can only be used on those types `t` which satisfy the `Num t` constraint. This `Num t` constraint is declared by the programmer as a typeclass. The instances of the typeclass specify when the typeclass constraint holds, and what is the intended behavior at that type. For example, the `Num` typeclass and its instances are shown in the Figure 1. The use of the function (+), say for example in the definition of `addF`, the typechecker needs to justify its type correctness. This amounts to satisfying the constraint `Num Float`. which in turn can only be satisfied, if the typechecker can find a declaration of the the instance declaration `Num Float`.

The idea of typeclasses originates from Wadler and Blott [1989] which modeled typeclass instances as dictonaries. The implicit class constraints—such as `Num t`—are elaborated into special data called dictonaries [Hall et al., 1994] internally by the compiler. Mathematically, typeclasses can be viewed as relations over types [Morris, 2014]. Every typeclass instance declaration extends that relation. The typeclass `Num` represents a unary relation for those types whose values behave like numbers, they can be compared (≤, ==) and added together (+). The declarations in the Figure 1 can be interpreted as an evidence of fact that `Float` and `Int` belong to the unary relation `Num`, { `Int`, `Float` } ⊆ `Num`.

3 Program Data

3.1 Algebraic Datatypes

The other important aspect of any programming language, the first being transformations of data described in the previous section, is that of organizing related data in a logical fashion for the ease of access and update. The data structures store the information by encoding the domain elements via an appropriate representation in memory. The transformation on these structures models the program logic. Algebraic datatypes (ADT) are a primary way to define such new structures in a functional programming language. They are called algebraic because they can be viewed as composite datatype of (named) sums of products. As an example, consider modeling a simple calculator application which performs addition operation on integers. The domain, in this case, are arithmetic expressions and can be represented using an ADT in Haskell as:

```
data AlgExp where
   Value :: Int    → AlgExp
   Plus  :: AlgExp → AlgExp → AlgExp

eval :: AlgExp → Int
eval (Value i) = i
eval (Plus x y) = eval x + eval y
```

Fig. 2. AlgExp and its evaluation

The **data** keyword defines a new user defined datatype with name AlgExp as shown in Figure 2. The data constructor Value stores Int values, while Plus encodes addition of two AlgExp expressions. The function of the calculator—to compute expressions—is simulated by evaluating the encoded expression performed by an eval function defined recursively.

The declarative style of programming allows us to write the eval function on a case by case basis via pattern matching. There are exactly two ways in which we could have obtained a value of type AlgExp. The first case says that if we have a Value i, we can deduce that i is an Int and return it. In the second case, Plus, we first evaluate the expressions x and y recursively, and then return the addition of the two results. Another advantage of having a declarative style is that extending the calculator application, say to include operations such as multiplication and division operation, is straightforward. The required code changes would be to add the associated data constructors, Mult and Div, in the datatype declaration followed by extending the eval function with cases for Mult, which multiplies, and Div that divides. In a typed setting the typechecker can identify and reject programs with ill-structured data. The pattern expression eval (Plus x) is ill-typed as Plus requires 2 arguments of type AlgExp.

The declarative style of defining algebraic datatypes and pattern matching was introduced in Hope [Burstall, 1969, Burstall et al., 1980] and has become an essential feature of all modern typed functional programming languages.

3.2 Newtypes

A special case of algebraic datatype is when there is only one data constructor for the user defined datatype. This wrapping of an existing type is to achieve data hiding. Newtype provides the mechanism which confines the visibility of the representation details of a type exclusively within a module. External to the module type representation is opaque to the client module. Consider an HTML module as shown in the Figure 3.

```
module HTML (Html, mkHtml, unMkHtml)
where

newtype Html = Html String

mkHtml :: String → HTML
mkHtml = Html ∘ escapeString

unMkHtml :: HTML → String
unMkHtml (Html s) = s
```

```
module Client
where

import HTML

page :: HTML
page = mkHTML "<html><body>Text</body></html>"
```

Fig. 3. HTML as a generative type

Within the scope of Html module, the types String and HTML are synonymous, however the type Html will be opaque for any external client using the type HTML. The Client module will not be able to directly manipulate a value of type

`Html`—unless of course by using the specific functions exposed in the signature. Inhibiting manipulations and views of the data representation is useful to avoid leaking information in secure computation contexts. A naive abstraction, however, will incur a runtime cost. `Html` and `String`, need to be explicitly converted from one form into another using pattern matching. This explicit conversion is redundant as they have the same representation in memory, but the compiler cannot use this information to perform any optimization on the generated code. Is it possible for the runtime semantics of the code be liberated from the extra overhead?

The idea of newtypes can be traced back to generative abstract types, which first appeared in work of Leroy [1995] and Milner et al. [1997] in the context of ML module systems. Generative abstract types in the context of Haskell are similar to [Montagu and Rémy, 1 21]. The idea of abstract types is even older due to TODO [2050]

3.3 Generalized Algebraic Datatypes

Consider extending the previously defined `AlgExp` from Figure 2 to also include a data constructor `IsZero`. The intention of the constructor is to encode the check if the expression of type `AlgExp` is evaluated to zero.

```
data AlgExp where                                    eval :: AlgExp → Int
  Value  :: Int             → AlgExp                 eval (Value i) = i
  Plus   :: AlgExp → AlgExp → AlgExp                 eval (Plus x y) = (eval x) + (eval y)
  IsZero :: AlgExp          → AlgExp                 eval (IsZero x) = (eval x) == 0 -- type error!
```

Fig. 4. Extending the `AlgExp` datatype and `eval` function

How should the `eval` function handle the `IsZero` case? The naive definition, `(eval x) == 0` as shown in Figure 4, fails to type check as the `eval` function requires the return type of the expression to be an `Int` but it is of type `Bool`. A possible solution is to change the return type of the the `eval` function to be `Either Int Bool`. This would mean that the evaluator either returns an `Int` or a `Bool`. While this works as expected, it requires a complete rewrite of the `eval` function. The situation of maintaining this `eval` function becomes even more cumbersome if later we wanted to add a new facility, say, to store and use user defined functions. Extending `AlgExp` with new constructs would mean nesting of `Either` datatype and then checking at each recursive call the value returned is the one that we expect as shown below:

```
                                   eval :: AlgExp → Either Bool Int
                                   eval (Value i) = Right i
                                   eval (Plus x y) =
data Either a b where                case (eval x, eval y) of
    Left  :: a  → Either a b           (Right x', Right y') → Right (x' + y')
    Right :: b  → Either a b           _                    → error "should not happen"
                                   eval (IsZero x) = case (eval x) of
                                                       Left i → Left (i == 0)
                                                       _      → error "should not happen"
```

To reflect on the situation: the programmer expects the `eval` function to compositionally evaluate an expression, then why should they be bothered to wrap the evaluated values of the `AlgExp` into an `Either` datatype? There needs to be an appropriate abstraction to handle this wrapping and unwrapping automatically. It is important to note that, in full generality, the evaluation of the expression may not result in the same type. In our case, the `IsZero` evaluates an

expression to a Boolean value, while the `Plus` operator evaluates to an Integer. Thus, the type signature of the `eval` function should abstract over the expression result type: `eval :: ∀ a. AlgExp a → a`. The problem would resolve if we were to constrain the return type of each of the data constructors to agree to the type of the value expected after evaluating the value of type `AlgExp`. This constrained return type can be used as a tag to convince the typechecker that `eval` function is indeed type safe. This formulation is shown in Figure 5 with the datatype `GAlgExp` along with the `eval` function which evaluates it.

```
data GAlgExp a where
  Value  :: Int                          → GAlgExp Int
  Plus   :: GAlgExp Int → GAlgExp Int → GAlgExp Int
  IsZero :: GAlgExp Int                  → GAlgExp Bool
```

```
eval :: GAlgExp a → a
eval (Value i) = i                      --(1)
eval (Plus x y) = (eval x) + (eval y)   --(2)
eval (IsZero x) = (eval x) == 0         --(3)
                    -- typechecks okay!
```

Fig. 5. `GAlgExp` datatype and `eval` function

In each of the case alternative branches, the type of the right hand side of a pattern match agrees with the type constraint introduced by the pattern. For example, in the alternative case branch labeled (2) in the Figure 5, the pattern `Plus x y` is of type `GAlgExp Int` and the type of the resultant on the right hand side of the equation is of type `Int`. Similarly, in the alternative branch labeled (3) with the pattern `IsZero x` is of type `AlgExp Bool` is on the left hand of the equation, has a right hand side of the equation of type `Bool`, which agrees with the type of its right hand side.

GADTs can further help in identifying dead code and code optimizations [Graf et al., 2020, Nilsson, 2005, Xi, 1998]. For example, consider the function `isZero` shown below:

```
isZero :: AlgExp → Bool
isZero (Value i) = i == 0
isZero (Plus x y) = isZero x && isZero y
isZero (IsZero x) = error "impossible"
```

```
isZero :: GAlgExp Int → Bool
isZero (Value i) = i == 0
isZero (Plus x y) = isZero x && isZero y
```

The typechecker has the necessary information to identify that the case `IsZero x` is impossible as it is of type `GAlgExp Bool` while the function only expects an argument of type `GAlgExp Int`. If we were to define the same function for `AlgExp`, the function would have to include an impossible case for `IsZero x`.

```
data GAlgExp a where
  ...
  Equals :: GAlgExp a → GAlgExp a → GAlgExp Bool
```

```
eval :: GAlgExp a → a
...
eval (Equals x y) = (eval x) == (eval y)
                                -- Type error!
```

Consider a generalization of `IsZero x` namely, `Equals x y` as shown in the above code block. The intention of `Equals x y` to represent comparing the arguments `x` and `y` and decide if they are equal. The `eval` function case that evaluates `Equals x y` fails to type check. The problem is not that the return type does not match, it is indeed `Bool`, but there is no reason to believe that `eval x` and `eval y` can be compared using the `(==)` operator. The only information we can infer from the pattern match is that the type of `eval x` and `eval y` are of some generic type variable `a`. Such types are called existential as they do not appear in the return type. The arugments to `Equal` need to be only those that can be compared.

By constraining the type variable to only those which satisfy the `Eq` typeclass, we can convince the typechecker that the term `(eval x) == (eval y)` is well typed.

```
data GAlgExp a where                          eval :: GAlgExp a → a
...                                           ...
   Equals :: Eq a ⇒ GAlgExp a → GAlgExp a     eval (Equals x y) = (eval x) == (eval y)
                  → GAlgExp Bool                                          -- Okay!
```

To summarize GADTs generalize the notion of algebraic datatypes in two distinct ways by uniformly supporting:

(1) phantom or tagged types, where the return type of the data constructor is no longer a generic type variable and it is refined to be a fixed type;
(2) existential types, where it is possible for the type variables that appear in the types of the arguments to the data constructors to not appear in the return type.

While there is a rich literature of GADTs (inductive type families) for dependently typed languages [Dybjer, 1991, 1994], the idea of GADTs for non-dependently typed languages appeared under different names such as indexed types [Zenger, 1997], first class phantom types [Cheney and Hinze, 2003], guarded recursive datatypes [Xi et al., 2003], equality qualified types [Sheard and Pasalic, 2008]. GHC/Haskell was the only language compiler that supported GADTs [Peyton Jones et al., 2004] until recently OCaml 4.0 [Garrigue and Le Normand, 2011] and Scala 3 [Xu et al., 2021] started supporting them.

4 Type Computation

4.1 Multiparameter Typeclasses

While single parameter typeclasses model unary relations over types, multiparameter typeclasses generalize single parameter typeclasses by modeling n-ary relations over types. For example, Jones [2000] uses the typeclass `Con e c` to unify the treatment of different containers and their contained elements as shown in the Figure 6. The type variable `e` stands in for the element type of the container while `c` stands in for the container type. The behavior of the containers is captured by the typeclass methods: `empty`, which returns the empty container, and `insert`, which inserts the element of type `e` in to the container of type `c`. We can imagine having instances for such a typeclass as `Con Int [Int]` that says that the container list of integers has integers as its elements, `Con Word (Tree Chars)` that says that a container tree of `Char` contains `Words` as its elements. Viewing them as relations, we have { `(Int, [Int])`, `(Int, Tree Int)` } ∈ `Con`.

```
class Con e c                instance Con Int [Int]        instance Con Word (Tree Char)
where                        where                         where
  empty :: c                   empty = ...                   empty = ...
  insert :: e → c → c          insert = ...                  insert = ...
```

Fig. 6. Con typeclass and its instances

The use of `empty` in a polymorphic setting, however, leads to ambiguity during compilation. The type of empty is inferred as `empty` as `Con e c ⇒ c`. This type in ambiguous as a free type variable `e` which appears only in the constraint. Ambiguous types do not have well defined semantics as the compiler cannot resolve the typeclass constraint to a unique instance and hence cannot choose the right implementation to use. Further, if the intention of the programmer is to

allow only homogeneous containers, multiparameter typeclasses are insufficient. Consider the term `con3a` which claims to insert both an `Int` value and a `Char` value into the argument `con`.

```
con3a con = insert 3 (insert 'a' con)
```

The typechecker infers the type of `con3a` as `(Con Int c, Con Char c) ⇒ c → c`. However, a use of this term to a container which contains only `Int` or `Char` will raise a type error. If the intention of the programmer was to allow only homogeneous containers, it would be helpful for the definition of the term `con3a` to be rejected, rather than a type error to appear at its use site.

4.2 Functional Dependencies

The problem with multiparameter typeclasses is that there is a mismatch between the programmers intention and the expressivity of the typeclass machinery. Continuing with the above example, the programmers intention might be to have a functional relation between `c` and `e` of the typeclass `Con e c`: fixing the type parameter `c` also fixes the the type parameter `e`. In general, there can be arbitrary functional relation between the type parameters of the typeclass. As seen in the Figure 7, the new annotation `c ⇝ e` on the typeclass definition captures the functional dependency of

```
class Con e c | c ⇝ e where
  empty :: c
  insert :: e → c → c
```

Fig. 7. The Con Typeclass with Functional Dependency

the typeclass `Con e c`. It is read as: *for a given type parameter* `c`, *the type parameter* `e` *can be uniquely determined*. The parameter `c` is called the determiner, while the parameter `e` is called the determinant of the functional dependency. This extension is attractive as there is a minimal change to the typeclass syntax, and it is backwards compatible with no change to typeclass instance declarations. The typechecker needs to verify that the new instance declarations do not violate the functional dependency: we cannot allow both the typeclass instances, `Con Int [Int]` and `Con Float [Int]` to exist together.

Jones [2000] introduced functional dependencies as a conservative extension to the typeclass language feature. His inspiration stems from relational algebra. Later, Jones and Diatchki [2008] raise questions about language designs in presence of functional dependencies, answer some of them while also clarifying the misunderstandings which had seeped deep into the functional programming community.

4.3 Associated Types

Another way of introducing a functional relation on types, or simply type functions, is via associating the determinant of the functional dependency within the class definition. Using the previous `Con c e` example, if we have a functional dependency `c ⇝ e` then, a type function, `Elem c`, can be used where ever the type parameter `e` appears.

In the Figure 8, the typeclass `Con` needs only one parameter, `c`, for the container type, with an additional field for the type function `Elem` that depends on the type parameter `c` of the typeclass. As a cascading effect, the type signature of `insert`, now only depends on the type parameter `c`, and the `Elem c` stands for the element type of container. In the instance declaration, the `Elem [Int]` is mapped to `Int`, meaning, the element of the list of integers is of type `Int`. To circumvent

```
class Con c where
  type Elem c
  empty :: c
  insert :: Elem c → c → c

instance Con [Int] where
  type Elem [Int] = Int
  empty = ...
  insert = ...

instance Con (Tree Char) where
  type Elem (Tree Char) = Word
  empty = ...
  insert = ...
```

Fig. 8. Con Typeclass with an Associated Type

the problem of ambiguous types, the multiparameter typeclass can be rewritten by replacing all the determinants of the functional dependency by the associated type applied to the determiners.

Associated types were introduced by Chakravarty et al. [2005] as an alternative for functional dependencies because of a more direct syntax: functional programmers are accustomed to writing functions and not relations as in logic programming.

Part II: System $\mathbf{F}_C$

5 History and Motivation

Haskell [Marlow and Peyton Jones, 2010] is statically typed. This has practical and theoretical implications. In practice, only those terms whose types can be verified by the typechecker can be compiled to the core intermediate language, and in theory, terms without types have no meanings. The Glasgow Haskell Compiler (GHC) [The GHC Team, 2020], is the de facto Haskell language compiler. The former iterations of GHC, used System **F** [Girard et al., 1989, Reynolds, 1974] with extensions, as its core intermediate language. In the rest of this section, we will briefly see how System **F**, with extensions encoded the language features but was insufficient for encoding newtypes.

GHC is based on Hindley-Milner type system [Milner, 1978]. It can be viewed as a restricted version of System **F** where all the types are in prenex form: the type variables are universally quantified at the outermost level and there are no explicitly existentially quantified types. The advantage of this system is that the term language is expressive enough to allow programmers write non-trivial programs while not burdening them with annotating every term with an explicit type. There is a decidable and complete algorithm which computes a most general unique type for a term.

The basic language features can be encoded in System **F** [Burstall et al., 1980]. The encoding amounts to making the implicit type in the surface level language explicit in function definitions and function applications. For example, the polymorphic function definition `id` is elaborated to:

```
id :: ∀ t. t → t
id A x = x
```

The term `id 3` written by the programmer is elaborated to `id Int 3`. Typeclasses or implicit adhoc polymorphism, is encoded via means of special terms called dictionaries [Hall et al., 1994, Wadler and Blott, 1989]. The typeclass instances are dictionaries which contain a reference to actual the method implementations. Elaboration of implicit adhoc polymorphism amounts to finding the right dictionary to be applied to the function as an explicit argument in the elaborated version of the function application.

The data constructors introduced by user defined datatypes are treated as term constants. Consider a pathological datatype declaration as shown below:

```
data T a where MkT :: a → T a
```

The data constructor `MkT` is of type `∀ a. a → T a` in the core language. The programmer written term `MkT 42` for the type `T Int` is elaborated to `MkT Int 42`.

System $\mathbf{F}_C$ [Sulzmann et al., 2007] was designed as an alternative to having separate extensions for each language feature. The key insight is that each language feature is, in a sense, trying to encode some notion of type equality ($\tau \sim \sigma$). GADTs introduce type refinements, as seen in `eval` example, are type equality constraints exposed after a pattern is matched. Instances of associated types introduce a type equality between the instantiated associated type and the concrete type (`Elem [Int] ~ Int`). The newtype introduces type equalities between the new defined type and its representation (`Html ~ String`). Consider a the following declaration:

```
newtype Fun = MkFun (Fun → Fun)
```

This newtype definition could not be encoded in the intermediate System **F** without using adhoc machinery. However, in System $\mathbf{F}_C$ this is achieved by introducing a type equality axiom: `Fun` $\sim$ `Fun` $\rightarrow$ `Fun`

In the following sections we describe the meta-theory of the core intermediate language followed by details of how each of the language features GADTs (in the Section 7.1), newtypes (Section 7.2), and associated types (Section 7.3.1), and also a unique feature of the system, open type functions (Section 7.3.2) are encoded in System $\mathbf{F}_C$ using *coercions* and *coercion axioms* which embody the notion of type equality.

6 Syntax

$$
\begin{array}{llll}
\text{Type Variables} & \alpha, \beta, \gamma & \text{Type Constants} & T \\
\text{Term Variables} & x, y & \text{Type Functions} & F \\
\text{Coercion Vars} & c & \text{Indices} & i, n \in \mathbb{N}
\end{array}
$$

$$
\begin{array}{lll}
\text{Kinds} & \kappa ::= & \star \mid \kappa \rightarrow \kappa \mid \boxed{\sigma \sim \tau} \\
\text{Types} & \tau, \sigma ::= & \alpha \mid T \mid \tau \rightarrow \tau \mid \tau\ \tau \mid \forall \alpha{:}\kappa.\ \tau \mid \boxed{F_n\ \overline{\tau}} \mid \boxed{\gamma} \\
\boxed{\text{Coercions}} & \nu, \gamma ::= & c \mid \langle \tau \rangle \mid \mathbf{sym}\ \gamma \mid \nu \circ \gamma \mid \forall \alpha{:}\kappa.\ \gamma \mid \gamma @ \tau \mid \nu\ \gamma \mid \mathbf{left}\ \gamma \mid \mathbf{right}\ \gamma \\
\text{Types/Coercions} & \phi ::= & \tau \mid \gamma \\
\text{Patterns} & P ::= & H\ \overline{\beta{:}\kappa}\ \overline{x{:}\tau} \\
\text{Terms} & M, N ::= & x \mid \lambda x{:}\tau.\ M \mid M\ N \mid \Lambda \phi{:}\kappa.\ M \mid M\ \tau \mid H \mid \text{case } M \text{ of } \overline{P \rightarrow M} \mid \boxed{M \blacktriangleright \gamma} \\
\text{Typing Context} & \Gamma, \Delta ::= & \epsilon \mid \Gamma, x{:}\tau \mid \Gamma, H{:}\tau \mid \Gamma, F_n\ \overline{\tau}{:}\kappa \mid \Gamma, \alpha{:}\kappa \mid \Gamma, c{:}\tau \sim \sigma \\
\text{Substitutions} & \Omega ::= & \epsilon \mid \{\overline{\alpha \mapsto \tau}\}
\end{array}
$$

$$
\begin{array}{lll}
\text{Program} & P_{gm} ::= & \overline{D_{cl}}\ ;\ \overline{x = M} \\
\text{Data Declarations} & D_{cl} ::= & \mathbf{data}\ \ T{:}\overline{\kappa} \rightarrow \star\ \ \mathbf{where}\ \ \overline{C_{trs}(T)} \\
 & \mid & \mathbf{type}\ \ F : \overline{\kappa} \rightarrow \kappa \\
 & \mid & \mathbf{axiom}\ \ c\ \overline{\alpha{:}\kappa} : \sigma_1 \sim \sigma_2 \\
\text{Data Constructors} & C_{trs}(T) ::= & H : \forall \overline{\alpha{:}\kappa}.\ \forall \overline{\beta{:}\kappa'}.\ \overline{\sigma} \rightarrow T\overline{\alpha}
\end{array}
$$

Fig. 9. The Syntax of System $\mathbf{F}_C$

The complete syntax of System $\mathbf{F}_C$ is shown in Figure 9. The system is classified into three sub-languages or levels: (1) kinds, (2) types, and (3) terms. The kind language sits at the highest abstraction level. It consists of base kind ($*$), higher kinds ($\kappa \rightarrow \kappa$), and the novel construct for expressing type equality ($\tau \sim \sigma$). Kinds classify types. The type language contains type variables (α), type constructors T, function type $\tau \rightarrow \sigma$, and polytypes ($\forall \alpha.\ \tau$) which support polymorphic functions. The type constant ranges over builtin types such as `Int` and user defined types. The system allows defining higher kinded types and inductive algebraic datatypes such as `Lists` and `Trees`.

The type equality coercion, γ , is the novel type level construct. It is classified by kind level type equality predicate, $\tau \sim \sigma$. Coercions are first class at the level of types: they can be constructed, applied to and passed in as arguments, and also abstracted over by using the special coercion infrastructure. System $\mathbf{F}_C$ supports a coercion calculus where each syntactic construct corresponds to a logical equation between two types. In practice coercions are types, and ϕ ranges over both types and coercions. The type function constants F_n need to appear fully saturated in types to be well formed, $F_n\ \overline{\tau}$. The subscript n specifies the arity of the type function, but it is elided when the context makes it clear. Coercions can also be introduced as equality axioms which can be thought as a templatized type equation. For example,

an axiom $\forall\alpha.\ F\ a \sim a$ says that for all types, `F a` can be treated exactly as `a`, and vice versa. We distinguish user defined datatypes T from type function constructors F. We use a special syntax $\tau_1 \sim \tau_2 \Rightarrow \sigma$ to abreviate $\forall\_{:}\tau_1 \sim \tau_2.\ \sigma$, meaning the coercion variable does not occur in the type σ.

Finally, types classify the terms of the system. The novel term construct of the system is $M \blacktriangleright \gamma$. It denotes that if a term M is of type τ and we have a coercion, γ, which says that $\tau \sim \sigma$ then, the term, $M \blacktriangleright \gamma$, justifies treating M as if it has type σ. Alternatively, the type equality coercion can be thought of as: if γ justifies $\tau \sim \sigma$ and M is of type τ then, within a context that expects a term of type σ, the term $M \blacktriangleright \gamma$ can be freely used, without worrying that the term will get stuck or crash during runtime. The "will not crash" guarantee is called as the *type soundness property* of the system. The type coercions introduced in the system should be thought of as axiomatic type equality, similar to axiomatic functional extensionality. However, the type soundness argument of the system, as we will see later, does not rely on any specific semantic notion of type equality.

The system has the usual term constructs: variables 'x', abstractions '$\lambda x{:}\tau.\ M$' and applications '$M\ N$', type level abstraction '$\Lambda\alpha.\ M$' and type applications '$M\ \sigma$'. Declaration of algebraic datatypes introduces data constructors H, the types of which are of the form:

$$H{:}\forall\overline{\alpha{:}\kappa}.\ \forall\overline{\beta{:}\kappa}.\ \overline{\sigma} \to T\ \overline{\alpha}$$

Here, the type variables, $\overline{\alpha}$, appear in the same order as in the algebraic datatype declarations, $T\ \overline{\alpha}$. The type variables $\overline{\beta}$ are special; they do not appear in the return type, $\overline{\alpha} \cap \overline{\beta} = \emptyset$. This characterization of splitting the type variables into $\overline{\alpha}$ and $\overline{\beta}$ plays a crucial role in representing GADTs and existential types. The case M of $\overline{P \to N}$ discriminates on the shape of the term M and chooses one of the alternatives N after a successful match on the one of the patterns P.

6.1 Static Semantics

$$\boxed{\Gamma \vdash_c \gamma : \kappa}$$

$$(\text{CO-REFL})\ \frac{\Gamma \vdash_t \tau : \kappa}{\Gamma \vdash_c \langle\tau\rangle : \tau \sim \tau} \qquad (\text{CO-SYM})\ \frac{\Gamma \vdash_c \gamma : \tau \sim \sigma}{\Gamma \vdash_c \mathbf{sym}\ \gamma : \sigma \sim \tau} \qquad (\text{CO-TRANS})\ \frac{\Gamma \vdash_c \gamma_1 : \tau \sim \tau_2 \quad \Gamma \vdash_c \gamma_2 : \tau_2 \sim \sigma}{\Gamma \vdash_c \gamma_1 \circ \gamma_2 : \tau \sim \sigma}$$

$$(\text{CO-}\forall\ I)\ \frac{\Gamma, \alpha{:}\kappa \vdash_c \gamma : \tau_1 \sim \tau_2 \quad \alpha\#\Gamma}{\Gamma \vdash_c \forall\alpha{:}\kappa.\ \gamma : \forall\alpha{:}\kappa.\ \tau_1 \sim \forall\alpha{:}\kappa.\ \tau_2} \qquad (\text{CO-}\forall\ E)\ \frac{\Gamma \vdash_c \gamma : \forall\alpha.\ \tau_1 \sim \forall\beta.\ \tau_2 \quad \Omega_1 = \{\alpha \mapsto \sigma\} \quad \Omega_2 = \{\beta \mapsto \sigma\}}{\Gamma \vdash_c \gamma @ \sigma : \Omega_1\tau_1 \sim \Omega_2\tau_2}$$

$$(\text{CO-COMP})\ \frac{\Gamma \vdash_c \gamma_1 : \tau_1 \sim \tau_2 \quad \Gamma \vdash_c \gamma_2 : \sigma_1 \sim \sigma_2 \quad \Gamma \vdash_t \tau_i\,\sigma_i : \kappa}{\Gamma \vdash_c \gamma_1\,\gamma_2 : \tau_1\,\sigma_1 \sim \tau_2\,\sigma_2} \qquad (\text{CO-F-COMP})\ \frac{\overline{\Gamma \vdash_c \gamma : \sigma \sim \tau}^{\,n} \quad \Gamma \vdash_t F\,\overline{\sigma}^{\,n} : \kappa}{\Gamma \vdash_c F\,\overline{\gamma}^{\,n} : F\,\overline{\sigma}^{\,n} \sim F\,\overline{\tau}^{\,n}}$$

$$(\text{CO-LEFT})\ \frac{\Gamma \vdash_c \gamma : \tau_1\,\sigma_1 \sim \tau_2\,\sigma_2}{\Gamma \vdash_c \mathbf{left}\ \gamma : \tau_1 \sim \tau_2} \qquad (\text{CO-RIGHT})\ \frac{\Gamma \vdash_c \gamma : \tau_1\,\sigma_1 \sim \tau_2\,\sigma_2}{\Gamma \vdash_c \mathbf{right}\ \gamma : \sigma_1 \sim \sigma_2}$$

$$(\text{CO-VAR})\ \frac{c{:}\tau \sim \sigma \in \Gamma}{\Gamma \vdash_c c : \tau \sim \sigma}$$

Fig. 10. Coercion Kinding Judgements: Excerpt of Static Semantics of System $\mathbf{F}_C$

The judgements in Figure 10 formalize the intuitions of how coercions ought to behave. Coercions are types such that their kinds say which what types considered equal. The coercion kinding judgment $\boxed{\Gamma \vdash_c \gamma : \kappa}$ is read as, "under the

assumptions in Γ, the coercion γ is (provably) of kind κ". The kinding rules (CO-REFL), (CO-SYM) and (CO-TRANS) makes type equality an equivalence relation. The reflexivity construct '$\langle\tau\rangle$' says that the type τ is equal to itself. The construct '**sym** γ' flips the direction of equality while transitivity '$\gamma_1 \circ \gamma_2$' chains two coercions γ_1 and γ_2.

The rules (CO-$\forall$ E) and (CO-$\forall$ I) justifies coercions between polytypes and their instantiations. If two polytypes are equal, then their instantiations with equal types are also equal by the rule (CO-$\forall$ E). The substitutions Ω_1 and Ω_2 maps the free type variables α and β to type σ. If two types, with a free type variable, are equal then type abstraction on both the types yields equal polytypes by the rule (CO-$\forall$ I). A necessary condition to avoid variable capture is that $\alpha\#\Gamma$ meaning, the variable α is fresh in the the environment. The rule (CO-COMP) enables combining coercions for higher kinded types and enable reasoning of equalities between them and similarly the rule (CO-F-COMP) enables coercion lifting for fully saturated type functions.

$$\boxed{\Gamma \vdash_t \tau : \kappa}$$

$$(\text{TY-VAR})\ \frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash_t \alpha : \kappa} \qquad (\text{TY-APP})\ \frac{\Gamma \vdash_t \sigma : \kappa' \to \kappa \quad \Gamma \vdash_t \tau : \kappa'}{\Gamma \vdash_t \sigma\,\tau : \kappa}$$

$$(\text{TY-CON})\ \frac{T{:}\kappa \in \Gamma}{\Gamma \vdash_t T : \kappa} \qquad (\text{TY-FCON})\ \frac{F_n{:}\overline{\kappa}^n \to \kappa' \in \Gamma \quad \overline{\Gamma \vdash_t \sigma : \kappa}^n}{\Gamma \vdash_t F_n\overline{\sigma}^n : \kappa'} \qquad (\text{TY-ALL})\ \frac{\Gamma, \alpha{:}\kappa \vdash_t \sigma : \star \quad \alpha \notin dom(\Gamma)}{\Gamma \vdash_t \forall\alpha{:}\kappa.\ \sigma : \star}$$

Fig. 11. Kinding Judgments: Excerpt of Static Semantics of System $\mathbf{F}_C$

The kinding rules for types, $\boxed{\Gamma \vdash_t \tau : \kappa}$, which are not coercions, are listed in Figure 11. They are fairly standard in comparison to System **F**. The kinding judgment rules (TY-FCON) and (TY-CON) are distinct as only fully saturated type function constructors are valid types in the system. The impredicative nature of System $\mathbf{F}_C$ is evident from the rule (TY-ALL): type quantification is a closed operation over types in the universe of kinds. Polytypes—the types with explicit type abstraction—and monotypes—the types with no explicit type abstraction— both are of kind $*$.

To illustrate usefulness of coercion composition, consider a higher kinded algebraic type `Tree` and a coercion $\gamma{:}\sigma_1 \sim \sigma_2$, then using τ_1 and τ_2 to be equal to `Tree`, we have that $\langle\texttt{Tree}\rangle\,\gamma : \texttt{Tree}\,\sigma_1 \sim \texttt{Tree}\,\sigma_2$. On the other hand, if we have a coercion $\texttt{Tree}\,\sigma_1 \sim \texttt{Tree}\,\sigma_1$ then we can recover the coercion components using the (CO-LEFT) and (CO-RIGHT), for the higher kinded type and its arguments respectively. In full generality, the lifting operation is defined in Definition 1 and the property is formailzed in Lemma 2.

Definition 1 (Coercion Lifting)**.** Let $\Omega = \{\alpha \mapsto \gamma\}$

$$\begin{aligned}
\Omega c &:= c \\
\Omega T &:= T \\
\Omega\langle\beta\rangle &:= \texttt{if}\ \alpha = \beta\ \texttt{then}\ \gamma\ \texttt{else}\ \langle\beta\rangle \\
\Omega\textbf{sym}\ \nu &:= \textbf{sym}\ \Omega\nu \\
\Omega\nu \circ \nu' &= \{\alpha \mapsto \gamma\}\nu \circ \{\alpha \mapsto \gamma\}\nu' \\
\Omega\textbf{right}\ \nu &:= \textbf{right}\ \Omega\nu
\end{aligned}
\qquad
\begin{aligned}
\Omega\beta &:= \texttt{if}\ \alpha = \beta\ \texttt{then}\ \gamma\ \texttt{else}\beta \\
\Omega(\tau \to \sigma) &:= (\Omega\tau) \to (\Omega\sigma) \\
\Omega(\tau\,\sigma) &:= (\Omega\tau)\,(\Omega\sigma) \\
\Omega(\forall\beta{:}\kappa.\ \tau) &:= \forall\beta{:}\kappa.\ (\Omega\tau) \\
\Omega(\nu@\tau) &:= (\Omega\nu)@(\Omega\tau) \\
\Omega\textbf{left}\ \nu &:= \textbf{left}\ \Omega\nu \\
\Omega(F\,\overline{\tau}) &= F\,\Omega\overline{\tau}
\end{aligned}$$

Theorem 2 (Coercion Lifting Lemma). *If* $\Gamma, \alpha{:}\kappa' \vdash_t \phi : \kappa$, *where* α *is free in* ϕ *and does not appear free in* Γ, $\Gamma \vdash_c \gamma : \sigma_1 \sim \sigma_2$, *and* $\Gamma \vdash_t \sigma_i : \kappa'$ *then,* $\Gamma \vdash_c \{\alpha \mapsto \gamma\}\langle\phi\rangle : \{\alpha \mapsto \sigma_1\}\phi \sim \{\alpha \mapsto \sigma_2\}\phi$

Proof Sketch Lemma 2. Proof is by induction on the derivation of the well kinded type ϕ. In each of the four cases, whenever in the original derivation the rule (co-refl) was used, it is replaced by the derivation of $\Gamma \vdash_c \gamma : \sigma_1 \sim \sigma_2$. □

$$\boxed{\Gamma \vdash M : \tau}$$

$$(\text{var})\ \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad (\to\text{I})\ \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.\, M : \sigma \to \tau} \qquad (\to\text{E})\ \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}$$

$$(\forall\text{I})\ \frac{\Gamma, \alpha{:}\kappa \vdash M : \tau \quad \alpha \# \Gamma}{\Gamma \vdash \forall \alpha{:}\kappa.\, M : \tau} \qquad (\forall\text{E})\ \frac{\Gamma \vdash M : \forall \alpha{:}\kappa.\, \tau \quad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash M\,\sigma : \tau}$$

$$(\text{cast})\ \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash_c \gamma : \tau \sim \sigma}{\Gamma \vdash M \blacktriangleright \gamma : \sigma} \qquad (\text{case})\ \frac{\Gamma \vdash M : \sigma \quad \overline{\Gamma \vdash P \to N : \sigma \to \tau}}{\Gamma \vdash \text{case } M \text{ of } \overline{P \to N} : \tau}$$

$$\boxed{\Gamma \vdash P \to N : \tau \to \sigma}$$

$$(\text{alt})\ \frac{H{:}\forall \overline{\alpha{:}\kappa}.\ \forall \overline{\beta{:}\iota}.\ \overline{\sigma} \to T\overline{\alpha} \in \Gamma \quad \Omega = \{\overline{\alpha \mapsto \tau'}\} \quad \Gamma, \overline{\beta{:}\Omega\iota}, \overline{x{:}\Omega\sigma} \vdash N : \tau}{\Gamma \vdash H\,\overline{\beta{:}\Omega\kappa}\,\overline{x{:}\Omega\sigma} \to N : T\overline{\tau'} \to \tau}$$

Fig. 12. Typing Judgments: Excerpt of Static Semantics of System $\mathbf{F}_C$

A typing rule for terms, $\boxed{\Gamma \vdash M : \tau}$, is shown in Figure 12. The rules inherited from System **F** are (var), (→I), (→E), (∀I), and (∀E). The novel rule (cast) transforms a term $M{:}\tau$ to a term $M \blacktriangleright \gamma : \sigma$ with a witness coercion $\gamma{:}\tau \sim \sigma$.

The two typing rules worth discussing are the ones for typing case statements (ty-case) and typing alternatives (ty-alt). A case statement can only be well typed if the discriminant is of type σ and each of the alternatives $P \to N$ have the same type $\sigma \to \tau$. For alternatives, if the pattern is a data constructor, only the existential type variables $\overline{\beta}$ are brought into scope explicitly in the pattern. The reason to do this is to ensure that the continuation term, N, can use the existentially bounded variables. However, we should be careful to not let the existential variables escape their scope. This invariant also needs to be enforced by the elaboration step from surface syntax into the core language. For example, consider the data constructor `IsEquals` and its type with explicit kind annotations from the previous section.

```
IsEquals :: ∀ (a :: ∗). ∀ (b :: ∗). DEq b → b → b → GAlgExp a
```

Inside a case term the alternative has the form

```
(IsEquals (b :: ∗). (d :: Eq b) (x :: b) (y :: b)) → N
```

where, N is the continuation term if the discriminant matches the pattern `IsEquals`, and the existential variable `d` the dictionary for the `Eq` typeclass, instances of which implements equality behavior over types. This type transformation is sound as the type of `IsEquals` is isomorphic to the following type:

```
∀ (a :: ∗). (∃ (b :: ∗). (Eq b, b, b)) → GAlgExp a
```

$$\begin{array}{lll} \text{Plain Values} & \mathcal{V} & ::= H \mid \lambda x{:}\tau.\, M \mid \Lambda\alpha{:}\kappa.\, M \\ \text{CValues} & C\mathcal{V} & ::= \mathcal{V} \mid \mathcal{V} \blacktriangleright \gamma \end{array}$$

6.2 Operational Semantics

The operational semantics formalizes the runtime behavior of the terms of the language. The value terms fall into two categories: plain values, $\mathcal{V}$, or coercion values, $C\mathcal{V}$, which are values with coercion casts. Value terms denote the "good" terms of the language which do not evaluate. For example, `True`, of type `Bool` is a value term. Coercion values, or cvalues, are needed to maintain the type preservation property. The terms involving coercions can step in one of the four interesting ways given by the rules (PUSH), (TY-PUSH), (CO-PUSH), or (H-PUSH) shown in the Figure 13.

$$\boxed{M \rightsquigarrow N}$$

$$(\text{PUSH})\ \frac{\gamma : \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2' \quad \gamma_1 = \mathbf{right}\ (\mathbf{left}\ \gamma) \quad \gamma_2 = \mathbf{right}\ \gamma}{((\lambda x.\, M) \blacktriangleright \gamma)\ N \rightsquigarrow ((\lambda x.\, M)\ (N \blacktriangleright \mathbf{sym}\, \gamma_1)) \blacktriangleright \gamma_2} \qquad (\text{TY-PUSH})\ \frac{}{(\Lambda\alpha{:}\kappa.\, M \blacktriangleright \gamma)\ \tau \rightsquigarrow (\Lambda\alpha{:}\kappa.\, (M \blacktriangleright \gamma @ \alpha))\ \tau}$$

$$(\text{CO-PUSH})\ \frac{\begin{array}{c} \nu{:}\sigma_1' \sim \sigma_2' \qquad \gamma{:}(\sigma_1 \sim \sigma_2 \Rightarrow \sigma_3) \sim (\sigma_1' \sim \sigma_2' \Rightarrow \sigma_3') \\ \gamma_1 : \sigma_1 \sim \sigma_1' = \mathbf{left}\ (\mathbf{left}\ \gamma) \quad \gamma_2 : \sigma_2 \sim \sigma_2' = \mathbf{right}\ (\mathbf{left}\ \gamma) \quad \gamma_3 : \sigma_3 \sim \sigma_3' = \mathbf{right}\ \gamma \end{array}}{(\Lambda\alpha{:}(\sigma_1 \sim \sigma_2).\, M \blacktriangleright \gamma)\ \nu \rightsquigarrow (\Lambda\alpha{:}(\sigma_1 \sim \sigma_2).\, M)\ (\gamma_1 \circ \nu \circ \mathbf{sym}\, \gamma_2) \blacktriangleright \gamma_3}$$

$$(\text{H-PUSH})\ \frac{\begin{array}{c} \gamma : T\,\overline{\sigma} \sim T\,\overline{\tau} \qquad \Omega = \{\overline{\alpha_i \mapsto \gamma_i}, \overline{\beta_i \mapsto \phi_i}\} \qquad \phi' = \begin{cases} \phi_i \blacktriangleright \Omega(\nu_1 \sim \nu_2) & \text{if } \beta_i : \nu_1 \sim \nu_2 \\ \phi_i & \text{otherwise} \end{cases} \\ H : \forall\overline{\alpha{:}\kappa}.\, \forall\overline{\beta{:}\kappa'}.\, \overline{\rho} \to T\,\overline{\alpha}^n \qquad M_i' = M_i \blacktriangleright \Omega\rho_i \\ \gamma_i = \mathbf{right}\ (\mathbf{left}^{\,n-i}\gamma) \end{array}}{\mathsf{case}\ (H\,\overline{\sigma}\,\overline{\phi}\,\overline{M} \blacktriangleright \gamma)\ \mathsf{of}\ \overline{\mathsf{P} \to N} \rightsquigarrow \mathsf{case}\ (H\,\overline{\tau}\,\overline{\phi'}\,\overline{M'})\ \mathsf{of}\ \overline{\mathsf{P} \to N}}$$

$$(\beta)\ \frac{}{(\lambda x{:}\tau.\, M)\ N \rightsquigarrow \{x \mapsto N\}M} \qquad (\text{Ty-}\beta)\ \frac{}{(\Lambda\alpha.\, M)\ \tau \rightsquigarrow \{\alpha \mapsto \tau\}M}$$

$$(\text{case})\ \frac{}{\mathsf{case}\ (H\,\overline{\sigma}\,\overline{\phi}\,\overline{M})\ \mathsf{of}\ \{\ldots; H\,\overline{\beta}\,\overline{x} \to N; \ldots\} \rightsquigarrow \{\overline{\beta \mapsto \phi}, \overline{x \mapsto M}\}N} \qquad (\text{Co-Trans})\ \frac{}{(\mathcal{V} \blacktriangleright \gamma) \blacktriangleright \nu \rightsquigarrow \mathcal{V} \blacktriangleright (\gamma \circ \nu)}$$

Fig. 13. Operational Semantics of System $\mathbf{F}_C$

In the rule (PUSH), the coercion γ, which is applied to the lambda term, is decomposed into two coercions; the first coercion, 'γ_1', is used in the coercion applied to the argument, the second coercion 'γ_2', is used to apply the complete term. Applying this transformation rule exposes a (β)-redux. This shows that coercions do not interfere with the function applications. The rule (TY-PUSH) for type application moves the coercion inside a type abstraction instantiated at the type variable. The rule (CO-PUSH) is just like (PUSH) but for moving coercions inside a coercion abstraction. Just like in the rule (PUSH), we see that the rule (TY-PUSH) enables evaluation for type functions applied to a coercion. The rule (CO-PUSH) is similar to the rule (TY-PUSH) but works on coercion abstractions. The type and coercion calculation ensures that the types are preserved when we push the coercions inside the body of a lambda bound term.

The most complex rule, (H-PUSH), pushes the coercion within the case term scrutinee. This rule is best illustrated with an example. Consider the term:

```
Cons (T Bool) x xs ▶ c
```

where `T` is a type constant, `Cons : ∀ a. a → List a → List a`, and `c : List (T Bool) ~ List Int`. The cast transforms the term from type `List (T Bool)` to type `List Int` while pushing the coercion into its sub-components with $\Omega = \{a \mapsto \textbf{right}\ c\}$

$$(\texttt{Cons}\ (\texttt{T}\ \texttt{Bool})\ \texttt{x}\ \texttt{xs}) \blacktriangleright c \rightsquigarrow \texttt{Cons}\ \texttt{Int}\ (\texttt{x} \blacktriangleright \textbf{right}\ c)(\texttt{xs} \blacktriangleright \langle \texttt{List} \rangle\ (\textbf{right}\ c))$$

After this transformation, the term is no longer is a coercion term, but a data constructor. If the original term was the case scrutinee, then the rule (CASE) can be applied only after the coercion has been pushed in to make progress in evaluation. The notation $\textbf{left}^{\,k}\gamma$ stands for **left** applied k times to γ. Coercion lifting plays an important role: it ensures that the term sub-components $M_i \blacktriangleright \Omega\rho_i$ are of the appropriate type. Finally, the rule (CO-TRANS) flattens a chain of two coercions, γ and ν, into a casted term with one coercion, $\gamma \circ \nu$.

The other rules, (β), (TY-β) and (CASE) are carried over from System **F** [Pierce, 2002]. As a short hand to chain multiple rules of the operational semantics defined in Figure 13, we define a multi-step reduction relation: $\bullet \rightsquigarrow^* \bullet$. It is a reflexive transitive closure of the single step operational semantics, $M \rightsquigarrow N$.

The operational calculus of coercions is not required during runtime. They are required to prove the soundness property of the calculus.

6.3 Meta-theory

The above formalization of static and operational semantics help us to answer the two important questions about the system:

(1) Do the static semantics effectively weed out all the programs that may fail at runtime?
(2) If we forget all the coercion and type annotations from the well typed term, can the same operational semantics be simulated?

The first question is answered by proving the syntactic soundness property [Wright and Felleisen, 1994], and the second, by showing that the system respects phase distinction [Harper et al., 1989].

6.3.1 Soundness.

Claim 1 (Progress and Subject Reduction). If $\Gamma \vdash M : \tau$ then, either $M \in C\mathcal{V}$ or, $M \rightsquigarrow M'$ and $\Gamma \vdash M' : \tau$

Informally, if we have a well typed term M, with type τ, then either the term is a value, and it cannot evaluate any further, or it can evaluate using one of the rules given by the operational semantics. Further, the type of the term never changes during evaluation. In practice, all those terms that may get a stuck are rejected by the typechecker as they will be ill-typed.

An important step while proving subject reduction is to ensure that anything obviously wrong such as `coBAD : Int ~ Bool` can never be derived in the system at the top level. Consider the term:

$$f = \Lambda c : \mathit{Int} \sim \mathit{Bool}.\ \mathit{not}\ (5 \blacktriangleright c)$$

with the environment $\Delta = \Gamma$, *not*:`Bool → Bool`. If it was possible to derive `coBad` using Δ then, f can be invoked, and it will crash the program. However, if the system prohibits derivation of such inconsistent coercions, f can never be

invoked. The Claim 1 thus needs to be strengthened using an appropriate restriction on Γ. System $\mathbf{F}_C$ uses a conservative check which is sufficient to ensure coercions like coBad can never be derived.

*Definition 3 (*Good Γ*).* A type environment, Γ, is Good Γ when it satisfies the following properties:

- If $\Gamma \vdash_c \gamma : T\overline{\sigma} \sim \tau$ and τ is a value type, then τ is of the form $T\,\overline{\sigma}'$
- If $\Gamma \vdash_c \gamma : (\sigma' \to \sigma) \sim \tau$ and τ is a value type, then τ is of the form $\tau' \to \tau''$
- If $\Gamma \vdash_c \gamma : \forall \alpha{:}\kappa.\ \sigma \sim \tau$ and τ is a value type, then τ is of the form $\forall \alpha{:}\kappa.\ \sigma'$

Where a type is a value type if it is of the form $T\overline{\tau}$, $\tau \to \sigma$, or $\forall \alpha{:}\kappa.\ \tau \sim \sigma$

Theorem 4 (Progress and Subject Reduction). *If* Good Γ *and* $\Gamma \vdash M : \tau$ *then, either* $M \in \mathcal{CV}$ *or,* $M \rightsquigarrow M'$ *and* $\Gamma \vdash M' : \tau$

Corollary 5 (Syntactic Soundness). *If* Good Γ *and* $\Gamma \vdash M : \tau$ *then, one of the cases holds:*

(1) $M \rightsquigarrow^* \mathcal{CV}$ *and* $\Gamma \vdash \mathcal{CV} : \tau$*, or;*
(2) evaluation of M *diverges*

6.3.2 Phase Distinction. The system also has the property where types do not interfere with the execution of the program. In other words, the types do not hold any computational significance, they are merely a compile time artifact. Even if all the type are erased from a well typed term, the term will not get stuck at runtime. Phase distinction is important as it provides formal guarantee that the efficient representation of the terms will not change its meaning. We formalize this in Corollary 8.

The erasure function, $|-|$, is defined on the term structure. It erases all the type and coercion annotations from a System $\mathbf{F}_C$ term and maps it to an "efficient" untyped lambda calculus term. The term, $()$, is a special constant. It contains no computational information and is used to simulate application of type level lambdas.

Definition 6 (Erasure of a System $\mathbf{F}_C$ Terms and Patterns)**.**
Erasure of terms $|\,M\,|$:

$$|\,x{:}\tau\,| = x \qquad |\,M \blacktriangleright \gamma\,| = |\,M\,|$$
$$|\,\lambda x{:}\tau.\,M\,| = \lambda x.\,|\,M\,| \qquad |\,M\,N\,| = |\,M\,|\,|\,N\,|$$
$$|\,\Lambda \alpha{:}\kappa.\,M\,| = \lambda \alpha.\,|\,M\,| \qquad |\,M\,\phi\,| = |\,M\,|\,()$$
$$|\,\mathsf{case}\ N\ \mathsf{of}\ \overline{\mathsf{P} \to M}\,| = \mathsf{case}\ |\,N\,|\ \mathsf{of}\ \overline{|\,\mathsf{P}\,| \to |\,M\,|} \qquad |\,H\,| = H$$

Erasure of Patterns $|\,\mathsf{P}\,|$:

$$|\,H\,\overline{\beta{:}\kappa}\,\overline{x{:}\tau}\,| = H\,\overline{\beta}\,\overline{x}$$

Theorem 7. *If* Good Γ *and* $\Gamma \vdash M : \tau$ *then one of the cases hold:*

(1) if M *is a* $\mathcal{CV}$ *then* $|\,M\,|$ *is a* $\mathcal{V}$
(2) if $M \rightsquigarrow M'$ *then,* $|\,M\,| \rightsquigarrow |\,M'\,|$ *or* $|\,M\,| = |\,M'\,|$

Corollary 8 (Erasure Soundness). *If* Good Γ *and* $\Gamma \vdash M : \tau$ *then* $M \rightsquigarrow^* M'$ *if and only if* $|\,M\,| \rightsquigarrow^* |\,M'\,|$

7 Encoding Language Features in System $\mathbf{F}_C$

7.1 GADTs

Before formalizing how terms containting GADTs data constructors can be encoded into System $\mathbf{F}_C$, it is illustrative to see how the data constructors themselves are represented in System $\mathbf{F}_C$. Recall the definition of `GAlgExp a` from Figure 5, its System $\mathbf{F}_C$ encoding is shown in Figure 14. The type tag of the return type of each data constructor is generalized to a generic type variable, with an additional type equality constraint that the generic type variable is equal to the type tag. The type tag in the case of `Value` is `Int`. This is the "Henry Ford" encoding [Chapman et al., 2010]: the type of `Value 0` is `GAlgExp a` as long as `a` is `Int`.

```
data GAlgExp a where
  Value  :: Int          → GAlgExp Int
  Plus   :: GAlgExp Int
         → GAlgExp Int → GAlgExp Int
  IsZero :: GAlgExp Int
                        → GAlgExp Bool
```

```
Value :: ∀(a :: ⋆). a ~ Int ⇒ Int → GAlgExp a
Plus  :: ∀(a :: ⋆). a ~ Int ⇒ GAlgExp Int
      → GAlgExp Int → GAlgExp a
IsZero :: ∀(a :: ⋆). a ~ Bool ⇒ GAlgExp Int
       → GAlgExp a
```

Fig. 14. `GAlgExp` datatype (left) and and its System $\mathbf{F}_C$ elaboration (right)

Suppose the programmer writes the following term: `e = Value 0`. The elaboration of `e` in System $\mathbf{F}_C$ makes the types and coercions explicit in the term. Thus the term in System $\mathbf{F}_C$ will be `Value Int <Int> 0`. The coercion argument is instantiated with the unique coercion `<Int> :: Int ~ Int`.

Although System $\mathbf{F}_C$ does not distinguish between monotypes and polytypes, the surface level syntax does. In Figure 15, the constraints, C, are named type equality predicates. The monotypes, τ, can be constrained using these type equality constraints. The polytypes, π, are quantified constrained types.

The typing-cum-elaboration judgment $\boxed{C; \Gamma \vdash_{\mathcal{G}} M : \pi \Longmapsto M'}$ denotes that: given a well typed surface level term M with type π, it is elaborated to a term M' in System $\mathbf{F}_C$ under the constraints C and typing environment Γ. The key observation is that the type equality constraints, $\tau \sim \tau'$, are elaborated to coercions in System $\mathbf{F}_C$. The rules (G-VAR), (G-$\forall$I) and (G-$\forall$ E) are standard rules used for elaborating Hindley-Milner style system [Wadler and Blott, 1989] to System $\mathbf{F}$. The rules (G-C I) and (G-C E) reminiscent of elaborating typeclass constraints. However, unlike typeclasses, the equality constraints are elaborated to types (coercions) and not term level constructs (dictonaries). The complex looking rule (G-ALT) elaborates case statements into System $\mathbf{F}_C$. Each surface level data constructor is elaborated to the System $\mathbf{F}_C$ data constructor with explicit existential coercions as pattern variables. The rule (G-EQ) is the same as (CAST) rule. In the rules (G-EQ) and (G-C E), the coercion, γ, is built using the context C. Constructing the appropriate γ algorithmically is possible by using a unification algorithm based on Lassez et al. [1998].

We can now demystify the type magic that the `eval` function performed by elaborating the definition of the eval function into System $\mathbf{F}_C$ as shown in Figure 16.

In the case alternative for `Value`, the argument to the constructor is `Int`, It is casted to the return type, `a`, using the coercion **`sym`** `co :: Int ~ a`. In the case alternative of `IsZero`, an existential coercion, `co :: a ~ Bool`, is brought into scope. The coercion, `co`, refines the result type of the expression `(eval Int x == 0)` from the actual type `Bool` to the generic variable `a`. The `Plus` case can be read systematically: each argument, `x` and `y`, is first evaluated at type `Int`. The

$$\begin{array}{ll} \text{Constraints} & C ::= \epsilon \mid C, c{:}\tau \sim \tau' \\ \text{Monotypes} & \nu, \tau ::= \alpha \mid \tau \to \tau \mid T\,\overline{\tau} \\ \text{Constrained Types} & \eta ::= \tau \mid C \Rightarrow \eta \\ \text{Polytypes} & \pi ::= \eta \mid \forall \alpha.\ \pi \end{array}$$

$$\boxed{C; \Gamma \vdash_{\mathcal{G}} M : \pi \Longmapsto M'}$$

$$(\text{G-VAR})\ \frac{x{:}\pi \in \Gamma}{C; \Gamma \vdash_{\mathcal{G}} x : \pi \Longmapsto x} \qquad (\text{G-EQ})\ \frac{C; \Gamma \vdash_{\mathcal{G}} M : \tau \Longmapsto M' \quad C \vdash_c \gamma : \tau \sim \tau'}{C; \Gamma \vdash_{\mathcal{G}} M : \tau' \Longmapsto M' \blacktriangleright \gamma}$$

$$(\text{G-}\forall\ I)\ \frac{C; \Gamma \vdash_{\mathcal{G}} M : \pi \Longmapsto M' \quad \alpha \# C, \Gamma}{C; \Gamma \vdash_{\mathcal{G}} M : \forall \alpha{:}\star\ .\ \pi \Longmapsto \Lambda \alpha{:}\star\ .\ M'} \qquad (\text{G-}\forall\ E)\ \frac{C; \Gamma \vdash_{\mathcal{G}} M : \forall \alpha{:}\star\ .\ \pi \Longmapsto M'}{C; \Gamma \vdash_{\mathcal{G}} M : \{\alpha \mapsto \tau\}\pi \Longmapsto M'\ \tau}$$

$$(\text{G-}C\ I)\ \frac{C, c : \tau \sim \tau'; \Gamma \vdash_{\mathcal{G}} M : \eta \Longmapsto M'}{C; \Gamma \vdash_{\mathcal{G}} M : \tau \sim \tau' \Rightarrow \eta \Longmapsto \Lambda(c{:}\tau \sim \tau').\ M'} \qquad (\text{G-}C\ E)\ \frac{C; \Gamma \vdash_{\mathcal{G}} M : \tau \sim \tau' \Rightarrow \eta \Longmapsto M' \quad C \vdash_c \gamma : \tau \sim \tau'}{C; \Gamma \vdash_{\mathcal{G}} M : \eta \Longmapsto M'\ \gamma}$$

$$\boxed{C; \Gamma \vdash_{\mathcal{G}} p \to M : \pi \to \pi \Longmapsto p' \to e'}$$

$$(\text{G-ALT})\ \frac{\begin{array}{c} H{:}\forall \overline{\alpha}.\ \forall \overline{\beta}.\ \overline{\tau' \sim \tau''} \Rightarrow \overline{\tau} \to T\overline{\alpha} \quad \overline{\alpha} \cap \overline{\beta} = \emptyset \quad \mathtt{fvs}(\overline{\tau}, \overline{\tau'}, \overline{\tau''}) = \mathtt{fvs}(\overline{\alpha}, \overline{\beta}) \quad \Omega = \{\overline{\alpha \mapsto \nu}\} \quad \overline{c} \# C, \Gamma \\ C, \overline{c{:}\Omega\tau' \sim \Omega\tau''}\ ;\ \Gamma, \overline{x{:}\Omega\tau} \vdash_{\mathcal{G}} M : \tau' \Longmapsto M' \end{array}}{C; \Gamma \vdash_{\mathcal{G}} H\,\overline{x} \to M : T\,\overline{\nu} \to \tau' \Longmapsto H\ (\overline{\beta{:}\star})\ (\overline{c{:}\Omega\tau' \sim \Omega\tau''})\ (\overline{x{:}\Omega\tau}) \to M'}$$

Fig. 15. Type Syntax and Type-directed Translation of GADTs in System $\mathbf{F}_C$

```
eval :: GAlgExp a → a
eval (Value i) = i
eval (Plus x y)
   = (eval x) + (eval y)
eval (IsZero x)
   = eval x == 0
```

```
eval :: ∀ a. GAlgExp a → a
eval a (Value Int co x) = x ▶ sym co
eval a (Plus Int co x y)
   = ((eval Int x) + eval Int y) ▶ sym co
eval a (IsZero Bool co x)
   = (eval Int x == 0) ▶ sym co
```

Fig. 16. The eval function (left) and its elaboration in System $\mathbf{F}_C$ (right)

sum of the evaulations is returned after casting it with the coercion, `sym co`, to match it with the return type of the function.

We want to ensure that the subject reduction property proved in Theorem 4 holds with this extension. The following lemmas are extensions of the meta-theoretic properties discussed in the previous section.

Lemma 9 (Type Preservation). *If* $C; \epsilon \vdash_{\mathcal{G}} M : \pi \Longmapsto M'$ *then* $C; \epsilon \vdash M' : \pi$

Type preservation is an important sanity check: the elaboration of a closed term does not change its type.

Theorem 10 (GADT Consistency). *If* Good Γ *and dom*(Γ) *does not contain type variables and coercion constants, and* $\Gamma \vdash_c \gamma : \tau \sim \tau'$ *then,* τ *and* τ' *are syntactically identical.*

Theorem 10 says that if two base types, or type function free types, are provably equal, then they must be syntactically identical. This is an important property which says makes it possible to use coercions safely. For example, the value co in `Val Int co` from Figure 16, can only be a $\langle Int \rangle$, the trivial coercion construct.

Finally, all GADT programs are sound in System $\mathbf{F}_C$ due to Lemma 9, Theorem 10 and Corollary 8

Corollary 11 (GADT Soundness). *If* $\epsilon; \epsilon \vdash_{\mathcal{G}} M : \tau \Longmapsto M'$, *then* $M' \rightsquigarrow^* \mathcal{V}$ *iff* $\mid M \mid \rightsquigarrow^* \mathcal{V}$ *where* $\mathcal{V}$ *is a value of a ground type.*

7.2 Newtypes

Newtypes have a trivial translation into System $\mathbf{F}_C$. Each surface level type declaration gives rise to an coercion axiom. For example, consider the **`newtype`** `HTML` example from Figure 3. The axiom `coHtml` can be used freely to cast any term with a type `HTML` to be used in the context that expects a `String`. The attractiveness of this encoding is that it enables type safety at zero runtime cost.

```
newtype Html = MkHtml String                    coHtml :: Html ~ String

mkHtml :: String → Html                         mkHtml :: String → Html
mkHtml x = Html (escapeString x)                mkHtml x = (escapeString x) ▶ (sym coHTML)

unHtml :: Html → String                         unHtml :: Html → String
unHtml x = case x of HTML y → y                 unHtml x = x ▶ coHtml
```

Fig. 17. **`newtype`** `HTML` functions (left) and its elaboration in System $\mathbf{F}_C$ (right)

For example, the function `mkHTML` as shown in Figure 17, the elaborated System $\mathbf{F}_C$ code will have a type cast instead of explicit boxing with a data constructor. In the elaborated `unHTML` function, code would contain a type cast instead of explicit unboxing using a **`case`** statement as shown above. Further, due to coercion lifting, this zero cost abstraction can be factored through compositionally to complex data types as well: `List Html` is treated exactly as `List String` as the coercion `List coHtml` justifies it.

A a new category of datatypes, separate from vanilla data constructors, is needed to formalize the above elaboration procedure using a type directed translation. As shown in Figure 18, this new category of newtype datatypes, T_N, has only one associated data constructor D_N. It specifies the newtypes representation.

$$
\begin{array}{lll}
\multicolumn{3}{c}{\text{Newtype Type Constructors} \quad T_N} \\
\multicolumn{3}{c}{\text{Newtype Data Constructors} \quad D_N} \\
\text{Types} & \tau, \sigma ::= & \alpha \mid \cdots \mid T_N \\
\text{Newtype Declaration} & D_{cl} ::= & \textbf{newtype}\ T_N\ \overline{\alpha} = D_N\ \tau
\end{array}
$$

Fig. 18. Extension Syntax for **`newtype`** in System $\mathbf{F}_C$

The type directed translation of the surface level datatype definition is shown in Figure 19. The rule (NT-AX) generates an axiom along with the type and term constants.

$$(\text{NT-AX})\ \frac{}{\Gamma \vdash_N \texttt{newtype}\ T_N\ \overline{\alpha} = D_N\ \{x : \tau\} \Longmapsto \begin{array}{l}\Gamma, D_N : \tau \to T_N\ \overline{\alpha}, x : T_N \overline{\alpha} \to \tau, \\ \texttt{axiom}\ co T_N : \forall \overline{\alpha}.\ T_N\ \overline{\alpha} \sim \tau\end{array}}$$

Fig. 19. Newtype type directed translation

The optimization step, of removing packing and unpacking of values of newtype, cannot be performed on algebraic datatypes. The runtime cost characteristics of types declared using **`newtype`**s and **`data`** are different. Due to the newtype definition `Html` *is a* `String` and there is no runtime cost for coercing `Html` to `String` and vice versa. The coercion axiom introduced during the elaboration step of the newtype declaration makes this zero runtime cost cast possible. Similar to GADT syntactic soundness, we also have the syntactic soundness for newtypes:

Theorem 12 (New types Syntactic Soundness). *If* Good Γ *and* $\Gamma \vdash_N M : \tau \Longmapsto M'$ *then,* $M' \rightsquigarrow^* \mathcal{V}$ *iff* $\mid M \mid \rightsquigarrow^* \mathcal{V}$ *where* $\mathcal{V}$ *is a value of a ground type.*

7.3 Type Computation

7.3.1 Associated Types. Reconsider the `Con c` typeclass which captures the behavior of containers from Figure 8. The function `sumCon :: (Con c, Num (Elem c)) ⇒ c → Elem c` sums up all the elements of the collection `c`. The predicate `Num (Elem c)` asserts that the elements of the collection are number like. A reasonable use of `sumCon` would be at type `List Int`; a list of integers can be folded into an integer by using a repeated addition operation. This amounts to instantiating the type parameter `c` with `List Int` as shown in Figure 20.

```
sumCon :: ∀ c. (Con c, Num (Elem c))
       ⇒ c → Elem c
instance Con (List Int) where
  type Elem (List Int) = Int
  ...

sumList :: List Int → Int
sumList as = sumCon as
```

```
dnumInt :: Num Int
dconList :: Con (List Int)
coAx :: Elem (List Int) ~ Int

sumList :: List Int → Int
sumList as = sumCon (List Int) dconList
                    (dnumInt ▶ sym (<Num>coAx)) as
```

Fig. 20. Functions with Associated Types (right) and their elaboration in System $\mathbf{F}_C$ (left)

`dnumInt` is the instance dictonary for `Num Int` and `dconList` is the instance dictonary for `Con (List Int)`. Type equality axiom `coAx` is obtained due to the associated type instantiation. It asserts that `Elem (List Int)` is `Int`. This explicit type equality is used to justify calling `sumCon` with a `dnumInt` dictonary casted to type `Num (Elem Int)`.

Each class declaration introduces a new class predicate name in the type environment along with its method names. With associated types, a new "type function" name—`Elem c` in the case of `Con c`—is added to the type environment. Each instance introduces a new axiom into the typing environment. For the `Con c` typeclass the instance `Con (List Int)` introduces the coercion `CoCon : Elem (List Int) ~ Int`. In general, each instance generates an axiom of the form

$$\texttt{co} : \forall\ \overline{\alpha{:}\star}.\ \texttt{F}\ \sigma \sim \sigma'$$

Class Declarations	$C_{ls} ::=$ **class** $D\,\overline{\alpha}$ **where** $\overline{tsig}; \overline{vsig}$
Instance Declarations	$I_{nts} ::=$ **instance** $D\,\overline{\tau}$ **where** $\overline{tbnd}; \overline{vbnd}$
Associated types	$tsig ::=$ **type** τ
Method signatures	$vsig ::= x{:}\tau$
Associated type instance	$tbnd ::=$ **type** $\tau = \sigma$
Method bindings	$vbnd ::= x = M$

Fig. 21. Class and Associated Types Surface Syntax

where $\mathtt{fvs}(\sigma) = \overline{\alpha}$ and $\mathtt{fvs}(\sigma') \subseteq \overline{\alpha}$. These axioms can also be viewed as type rewrite inducing axioms as it rewrites the type $F\,\overline{\sigma}$ to the right hand side of the equation, σ. However, these type rewrite axioms are indeed type equivalences, as the typechecker can replace an occurance of σ' to $F\,\overline{\sigma}$

$$(\textsc{subst})\ \frac{C \vdash_{\mathcal{D}} D\,\tau \Mapsto d \quad C \vdash_c \gamma : D\,\tau \sim D\,\tau'}{C \vdash_{\mathcal{D}} D\,\tau' \Mapsto d \blacktriangleright \gamma}$$

To formalize the elaboration procedure, the typing rules (G-$C\ I$) and (G-$C\ E$) from Figure 15 are reused. They are now generalized where the constraint set C contain typeclass constraints along with type equality predicates. The rule (SUBST) captures the translation of casting dictonaries as seen in the above example. The judgment $C \vdash_{\mathcal{D}} D\,\tau \Mapsto d$ elaborates the predicate $D\,\tau$ to a dictionary d in System $\mathbf{F}_C$. In the type checker implementation, the typeclass constraint solver builds the dictonary evidence [Hall et al., 1994].

Theorem 13 (Associated Type Consistency). *If* Γ *contains type rewrite axioms that are confluent and terminating, then* Γ *is consistent.*

For non-overlapping typeclass instances, the rewrite axioms are indeed confluent and terminating. The soundness theorem for associate types as it is exactly the same as for GADT soundness.

7.3.2 Open Type Functions. The associated types can also be disassociated from their typeclasses and have an independent existence. Such types are called type functions or type families. They are open, meaning, just like class instances, they can be extended. An example of how type computations are expressed by type functions that define addition of naturals at type level, shown in Figure 22.

```
data Z
data S n
```

```
type family Plus m n
type instance Plus Z n = n
type instance Plus (S m) n = S (Plus m n)
```

Fig. 22. Type level Arithmetic

Each type family instance directly translates to coercion axioms. The two instances for the `Plus` type family introduces the following axioms:

```
coPlusZn :: ∀ n. Plus Z n ~ n
coPlusSmn :: ∀ m n. Plus (S m) n ~ S (Plus m n)
```

There are certain caveats on how these coercion axioms introduced by type functions can be used by the typechecker to avoid inconsistency. For example, consider an open type function `F` and two of its instances that give rise to axioms `coFIB` and `coFBB` respectively.

```
type family F a
type instance F Int = Bool
type instance F Bool = Bool
```

```
axiom coFIB: F Int ~ Bool
axiom coFBB: F Bool ~ Bool
```

We can derive a coercion `Int ~ Bool` if we are careless. Both axioms have `Bool` as their result type, and hence we can derive `F Int ~ F Bool` by transitivity, and then we can use the **`right`** construct to derive `Int ~ Bool`. The following term

```
right (coFIB ∘ sym coFBB)
```

has the type `Int ~ Bool`, a blunder! It is crucial to use the **`left`** and **`right`** rules only on types that are not applications of type functions as they can be non-injective. This also influences the surface language design: the type functions are special and need to be always fully saturated.

7.4 Observations

In the current formalization, all the types and coercions are explicit in the system. While this makes a machine implementation of the system easy to design and work with, it is difficult for humans to read and understand the System $\mathbf{F}_C$ terms. As an optimization step, it may be worth while to see if it is possible to elide some of the explicit coercions and then reconstruct them using a type synthesis algorithm. We first define a system with implicit coercions: System $\mathbf{F}_{C_i}$. The key difference between System $\mathbf{F}_C$ and System $\mathbf{F}_{C_i}$ is that where System $\mathbf{F}_C$ has a coercion type γ of kind $\tau \sim \tau'$, System $\mathbf{F}_{C_i}$ only gives the equality kind in curly braces $\tau \sim \tau'$. Hence, for terms

- Type casts, $M \blacktriangleright \gamma$, turns into $M \blacktriangleright \{\tau \sim \tau'\}$ and,
- Coercion applications, $M\,\gamma$ turns into $M\,\{\tau \sim \tau'\}$

Theorem 14 (Undecidability of coercion reconstruction of System $\mathbf{F}_{C_i}$). *If M_i is an expression in System $\mathbf{F}_{C_i}$ and Γ is a typing environment then, reconstructing a System $\mathbf{F}_C$ term M such that $\Gamma \vdash M_i \rightsquigarrow M : \sigma$, where $\Gamma \vdash M : \sigma$ holds is undecidable.*

The proof of undecidability amounts to reducing the problem of coercion reconstruction to A-ground theories in a semi-Thue system, which known to be undecidable [Post, 1947]. If there exists an alternative formulation of System $\mathbf{F}_{C_i}$ with fewer explicit type equalities and is sufficient to encode all the language features while enjoying decidable type checking is an open question. This result is not surprising as type synthesis problem for an arbitrary System **F** term is known to be an undecidable problem [Wells, 1999]. The intention of System $\mathbf{F}_C$ is to be used as internal language for the compiler. The users of the language would never be required to read the verbose code in the same way we do not expect programmers to understand x86 assembly code.

The second observation is that the coercions are types, kinds of which give type equalities. This characterization is novel to System $\mathbf{F}_C$. The status quo is to express coercions as terms [Baars and Swierstra, 2002, Neis et al., 2011, Sheard and Pasalic, 2008, Weirich, 2000]. The goal of the core language is to be practical; due to all coercions being encoded

at type level, they can be erased at runtime to generate efficient code. Another reason is that for implementation purposes, the error term, $\bot$ which trivially just halts the program, by throwing an exception, can be typed at all types. If equality had been encoded at the level of types by making coercions a term level entity, we would have an "error coercion" term, $\bot :: \tau \sim \sigma$. This would require guaranteeing that evaluating the type equality evidences does not lead to non-termination before we evaluate the term. Without this evaluation check we cannot use a coercion for casts while also guaranteeing type soundness. To avoid such complications and maintain a phase distinction System $\mathbf{F}_C$ follows the slogan: *Kinds are propositions for type equality, proofs are (coercion) types*

Part III: Extensions of System F_C

8 System F_C^R

```
newtype Html           = String
type family F a
type instance F Html   = Bool
type instance F String = Char
```

$$coHtml : \texttt{Html} \sim \texttt{String}$$
$$coFHS : \texttt{F Html} \sim \texttt{Bool}$$
$$coFSC : \texttt{F String} \sim \texttt{Char}$$

Fig. 23. Type Functions and Newtypes

Consider the previously defined newtype data definition: **newtype** `HTML`, and a hypothetical open type function, `F` as shown in Figure 23. The type family instance declarations add two new type equalities: `F Html ~ Bool` and `F String ~ Char`, along with the type equality `Html ~ String` due to newtype declaration. Now, using these three equalities we can derive a type unsound coercion `Char ~ Bool`. By virtue of the type function axiom we have `Char ~ F String`, then coercion lifting derives `F String ~ F Html` (as `String ~ Html`), and finally, obtain `Char ~ Bool` (by using `F Html ~ Bool`, and `F String ~ Char`):

$$\texttt{Char} \sim \texttt{F String} \sim F\,\texttt{Html} \sim Bool$$
$$(\textbf{sym}\ coFSC) \circ (F\ (\textbf{sym}\ coHtml)) \circ coFHS : Char \sim Bool$$

The reason for this unsound behaviour is the subtle interaction of the two features: type functions and newtypes. When viewed separately, as formulated in the previous section, the consistency criteria is sufficient to guarantee soundness. Weirich et al. [2011] gives similar type unsoundness behavior by using GADTs and newtypes. The crux of the problem is *unconstrained coercion lifting*: the use non-parametric features (type functions, GADTs) of the language in the context that assumes parametricity (newtypes). The key extension to System $\mathbf{F}_C$ is making the type equality finer grained. This enriches the type equality by encoding *how* the two types are equal. In the above case, we see that we have two flavors of type equality coercions: (i) nominal equality, which is established via type function instances (`F String ~ Char`), and (ii) representational equality which is established via newtype definitions (`Html ~ String`). The type system needs to reject unsound coercions like `F (sym coHtml) : F String ~ F Html`.

8.1 Syntax

The syntax of System $\mathbf{F}_C^R$ [Breitner et al., 2014, Weirich et al., 2011] in shown Figure 24. The highlighted portions are the extensions to System $\mathbf{F}_C$. The concept of finer grained type equality is captured by *roles*, ρ. Each type parameter to a type constant is decorated with a role along with its kind. The role annotations specify at what level the type parameter is to be compared. The convenience function `roles`$(\mathcal{T})$ returns the role of each of the type parameters of a type constant $\mathcal{T}$ as a sequence, $[\rho]$, which preserves the order in which it occurs in its definition.

System $\mathbf{F}_C^R$ has three different roles for three different flavors of equalities:

- **Nominal:** The strictest kind of equality denoted by $\sim_N$ which holds exactly when the two types are the *same*. For example, an type function axiom `F Int = Bool` makes `F int` $\sim_N$ `Bool`.

$$
\begin{array}{rr@{\qquad}rr}
\text{Type Vars} & \alpha, \beta, \gamma & \text{Type constants} & \mathcal{T} \\
\text{Term Vars} & x, y & \text{Newtypes} & T_N \\
\text{Coercion Vars} & c & \text{Type Function Names} & F_n \\
 & & \text{Indices} & i, n \in \mathbb{N}
\end{array}
$$

$$
\begin{array}{lrl}
\text{Roles} & \rho & ::= \ \mathtt{N} \mid \mathtt{R} \mid \mathtt{P} \\
\text{Kinds} & \kappa & ::= \ \star \mid \kappa \to \kappa \mid \sigma \sim_\rho \tau \\
\text{Types} & \tau, \sigma & ::= \ \alpha \mid \mathcal{T} \mid \tau\,\tau \mid \forall \alpha{:}\kappa.\ \tau \mid \gamma \\
\text{Type Constants} & \mathcal{T} & ::= \ T \mid (\to) \mid T_N \mid F_n \overline{\tau} \\
\text{Coercions} & \nu, \gamma & ::= \ c \mid \langle \tau \rangle \mid \mathbf{sym}\,\gamma \mid \nu \circ \gamma \mid \forall \alpha{:}\kappa.\ \gamma \mid \gamma @ \tau \mid \nu\,\gamma \\
 & & \quad \mid \mathbf{left}\,\gamma \mid \mathbf{right}\,\gamma \mid \mathbf{nth}^i\,\gamma \mid \mathcal{T}\,\overline{\gamma} \mid F\overline{\gamma} \mid \mathbf{sub}\,\gamma \\
\text{Types/Coercions} & \phi & ::= \ \tau \mid \gamma \\
\text{Patterns} & P & ::= \ H\,\overline{\beta{:}\kappa}\,\overline{x{:}\tau} \\
\text{Terms} & M, N & ::= \ x \mid \lambda x{:}\tau.\ M \mid M\,N \mid \Lambda \tau{:}\kappa.\ M \mid M\,\phi \mid H \mid \mathsf{case}\ M\ \mathsf{of}\ \overline{P \to M} \mid M \blacktriangleright \gamma \\
 & & \mathtt{roles}(\mathcal{T}) & ::= [\rho \mid \alpha{:}\rho, \alpha \in \mathtt{Params}(\mathcal{T})] \\
 & & \mathtt{Params}(T) & ::= [\alpha \mid T\,\overline{\alpha}{:}\star] \\
 & & \mathtt{Params}(T_N) & ::= [\alpha \mid T_N\,\overline{\alpha}{:}\star] \\
 & & \mathtt{Params}(F_n) & ::= [\alpha \mid F_n\,\overline{\alpha}^n]
\end{array}
$$

Fig. 24. Excerpt of Syntax of System $\mathbf{F}^R_C$; extension of System $\mathbf{F}_C$

- **Representational:** This equality holds when the two types have the same runtime representation. For example, a new type declarations **`newtype`** `Age = Int` makes $\mathtt{Age} \sim_R \mathtt{Int}$
- **Phantom:** The weakest kind of equality holds for all types: `a` $\sim_P$ `b` is valid at all types `a` and `b`

For example, consider the type definitions shown in Figure 25. In the case of `Maybe a`, the type variable `a` is used parametrically and hence has a role `R`. In a type function, `F a`, the type variable is used non-parametrically—the type function instances inspect the instantiated type value—and hence has a role `N`. In the case of `DT1 a` and `DT2 a` the type variable takes the role `N` as it occurs inside a type function, albeit deeply nested. Further, as expected, in case of `DT3 a`, the type variable is at role `R`, but in the case of `DT4 a`, the type variable is at role `N`. Although it is used at role `R` in the first argument to the constructor, it is used at role `N` in the second. The role of a parameter is the least upper bound of the role lattice due to the ordering `N` < `R` < `P`. Finally, the role of the parameter in `DT5` is `P`, as it plays no role in the structure of the type definition.

```
data Maybe a = Nothing | Just a                    roles(Maybe) = [ R ]
type family F a                                    roles(F)     = [ N ]

data DT1 a = MkDT1 (Maybe (F a))                   roles(DT1) = [ N ]
data DT2 a = MkDT2 (F (Maybe a))                   roles(DT2) = [ N ]
data DT3 a = MkDT3 (Maybe (Maybe a))               roles(DT3) = [ R ]
data DT4 a = MkDT4 (Maybe a) (F a)                 roles(DT4) = [ N ]
data DT5 a = MkDT5                                 roles(DT5) = [ P ]
```

Fig. 25. Type definitions and their respective roles

8.2 Static Semantics

The novel feature of System $\mathbf{F}_C^R$, "equality at role ρ", is captured by the coercion kinding judgements: $\Gamma \vdash_c \gamma : \tau \sim_\rho \sigma$. It is read as "in the type environment Γ the coercion γ, witness the equality between types τ and σ at role ρ".

$$\boxed{\Gamma \vdash_c \gamma : \tau \sim_\rho \sigma}$$

$$(\text{CO-REFL})\ \frac{\Gamma \vdash_t \tau : \kappa}{\Gamma \vdash_c \langle \tau \rangle : \tau \sim_N \tau} \qquad (\text{CO-TRANS})\ \frac{\Gamma \vdash_c \gamma : \tau \sim_\rho \tau' \quad \Gamma \vdash_c \gamma' : \tau' \sim_\rho \tau''}{\Gamma \vdash_c \gamma \circ \gamma' : \tau \sim_\rho \tau''} \qquad (\text{CO-SUB})\ \frac{\Gamma \vdash_c \gamma : \tau \sim_\rho \tau' \quad \rho < \rho'}{\Gamma \vdash_c \mathbf{sub}\, \gamma : \tau \sim_{\rho'} \tau'}$$

$$(\text{CO-NTH})\ \frac{\Gamma \vdash_c \gamma : T\,\overline{\sigma} \sim_R T\,\overline{\tau} \quad \overline{\rho} = \text{roles}(T)}{\Gamma \vdash_c \mathbf{nth}^i\, \gamma : \tau_1 \sim_{\rho_i} \tau_2} \qquad (\text{CO-LEFT})\ \frac{\Gamma \vdash_c \gamma : \tau_1\,\sigma_1 \sim_N \tau_2\,\sigma_2}{\Gamma \vdash_c \mathbf{left}\, \gamma : \tau_1 \sim_N \tau_2} \qquad (\text{CO-RIGHT})\ \frac{\Gamma \vdash_c \gamma : \tau_1\,\sigma_1 \sim_N \tau_2\,\sigma_2}{\Gamma \vdash_c \mathbf{right}\, \gamma : \sigma_1 \sim_N \sigma_2}$$

$$(\text{CO-TY-APP})\ \frac{\begin{array}{c}\Gamma \vdash \tau_1\,\sigma_1 : \kappa \quad \Gamma \vdash_c \gamma_1 : \tau_1 \sim_\rho \tau_2 \\ \Gamma \vdash \tau_2\,\sigma_2 : \kappa \quad \Gamma \vdash_c \gamma_2 : \sigma_1 \sim_N \sigma_2\end{array}}{\Gamma \vdash_c \gamma_1\,\gamma_2 : \tau_1\sigma_1 \sim_\rho \tau_1\sigma_1} \qquad (\text{CO-F-APP})\ \frac{F \in \Gamma \quad \Gamma \vdash F\,\overline{\tau} : \kappa \quad \Gamma \vdash F\,\overline{\sigma} : \kappa \quad \overline{\Gamma \vdash_c \gamma : \tau \sim_N \sigma}}{\Gamma \vdash_c F\overline{\gamma} : F\overline{\tau} \sim_N F\overline{\sigma}}$$

$$(\text{CO-TYCON-APP})\ \frac{\overline{\rho} = \text{roles}(\mathcal{T}) \quad \begin{array}{c}\Gamma \vdash \mathcal{T}\,\overline{\tau} : \kappa \\ \Gamma \vdash \mathcal{T}\,\overline{\sigma} : \kappa\end{array} \quad \overline{\Gamma \vdash_c \gamma : \tau \sim_\rho \sigma} \quad \mathcal{T} \in \{T, T_N\}}{\Gamma \vdash_c \mathcal{T}\,\overline{\gamma} : \mathcal{T}\,\overline{\tau} \sim_R \mathcal{T}\,\overline{\sigma}}$$

$$\boxed{\Gamma \vdash M : \tau}$$

$$(\text{CAST})\ \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash_c \gamma : \tau \sim_R \tau'}{\Gamma \vdash M \blacktriangleright \gamma : \tau'}$$

Fig. 26. Excerpt of Static Semantics of System $\mathbf{F}_C^R$: Coercion Calculus

If two types are nominally equal, they are representationally equal as well. Similarly, if the two types are representationally equal, they are also equal at phantom role. This is captured in the rule (CO-SUB). The rule (CO-REFL) says that a well formed (kinded) type gives rise to a nominal type equality. The rule (CO-TRANS) says that coercions can be chained only if they are at the same role level. The construct **sym** γ, has no effect on the role of the type equality of γ. The rule (CO-TYCON-APP) builds representational type equality for newtypes and datatypes while (CO-F-APP) builds a nominal type equality for type functions. The rule (CO-TY-APP) builds a coercion between two type applications only if the argument type, $\sigma_1 \sim_N \sigma_2$, is at nominal level. The rules (CO-LEFT) and (CO-RIGHT) work only on nominal type equality for type applications.

The rule (CO-NTH) is restricted to non-type function types, T and T_N. The application of (CO-NTH) type functions is unsound as knowing that $F_n\overline{\tau} \sim F_n\overline{\sigma}$ does not give any information about the relationship between $\overline{\tau}$ and $\overline{\sigma}$. Type functions are non-injective. Although newtypes are injective, using the rule (CO-NTH) still poses a threat to soundess. Consider the following definitions using `Any a` and `App f a`:

```
data Any a = Any
newtype App f a = MkApp (f a)
```

$$\texttt{roles\{Any\}} = [P]$$
$$\texttt{App (f a)} \sim_R \texttt{f a}$$
$$\texttt{roles\{App\}} = [R, N]$$

Now, due to the rules of the coercion calculus we can derive the following representational type equality:

$$\texttt{App}\ \ \texttt{Any Int} \sim_R \texttt{App Any}\ \mathit{Bool} \tag{app-co}$$

This is possible because of the following chain of reasoning:

$$\texttt{App Any Int} \sim_R \texttt{Any Int} \sim_R \texttt{Any Bool} \sim_R \texttt{App Any}\ \mathit{Bool}$$

The first step and the last step of the reasoning chain is valid due to the instantiations of the coercion axiom, $\texttt{coApp} : \forall f, a.\ .\texttt{App}\ f\ a \sim_R f\ a$. Using the rule (CO-NTH) will produce a unsound coercion $\texttt{Int} \sim_R \texttt{Bool}$.

Type safe casts are expressed using the rule (CAST). It is sufficient to say that the type equality is at representational role as the terms will have the same representation in memory at runtime and thus also have identical behavior.

There is no change in operational semantics. Specifically, the push rules from the previous section Figure 13 still hold.

8.3 Meta-theory

The proof of progress lemma does not change with respect to System $\mathbf{F}_C$ as the only additional construct is roles, which does not play any part in the operational semantics of the language. To recall, the proof of progress is necessary to show that the well typed terms are either values or they can take a step.

THEOREM 15 (PROGRESS FOR SYSTEM $\mathbf{F}_C^R$). *If* $\texttt{Good}\ \ \Gamma$ *and* $\Gamma \vdash M : \tau$ *then, either* $M \in \mathcal{CV}$ *or,* $M \rightsquigarrow N$

The proof of preservation uses the formulation of coercion calculus extended with roles. The reason for type unsoundness in System $\mathbf{F}_C$ was that we allowed too many coercions with unrestricted coercion lifting. The coercion lifting lemma in this system is stated below:

LEMMA 16 (SYSTEM $\mathbf{F}_C^R$ COERCION LIFTING). *If* $\Gamma, \alpha{:}\kappa' \vdash_t \phi : \kappa$, *where* α *is free in* ϕ *and does not appear free in* Γ, $\Gamma \vdash_c \gamma : \sigma_1 \sim_N \sigma_2$, *and* $\Gamma \vdash_t \sigma_i : \kappa'$ *then,* $\Gamma \vdash_c \{\alpha \mapsto \gamma\}\langle\phi\rangle : \{\alpha \mapsto \sigma_1\}\phi \sim_N \{\alpha \mapsto \sigma_2\}\phi$

As a consequence of the refined coercion lifting, the type system disallows the unsound coercions such as `F (sym coHtml) : F String` $\sim_N$ `F Html`. The type equality `String` $\sim_R$ `Html` cannot be lifted through the type equality `sym (F a) : F a` $\sim_N$ `F a`. Finally, types are invariant with respect to the operational semantics.

LEMMA 17 (TYPE PRESERVATION FOR SYSTEM $\mathbf{F}_C^R$). *If* $\texttt{Good}\ \ \Gamma$ *and* $\Gamma \vdash M : \tau$ *and* $M \rightsquigarrow M'$, *then* $\Gamma \vdash M' : \tau$

9 System $\mathbf{F}_C^\kappa$

```
data TyRep :: forall k. k → Type where
  TyInt  :: TyRep Int
  TyBool :: TyRep Bool
  TyList :: TyRep []
  TyApp  :: TyRep a → TyRep b → TyRep (a b)
```

```
baseElem :: forall (a :: Type). TyRep a → a
baseElem TyInt = 0
baseElem TyBool = False
baseElem (TyApp TyList _) = []
```

Fig. 27. Representing Higher Kinded Types

Programmers may want to encode complex constraints via GADTs. For example, consider the datatype declaration `TyRep` and the function `baseElem` which accepts `TyRep a` as shown in Figure 27.

In this definition of `TyRep`, the constructors `TyInt` and `TyBool` represent the base type `Int` and `Bool`, `TyList`, represents the list type, and `TyApp` represents type application. The `TyRep` datatype encodes only well kinded types. The `baseElem` function takes only those `TyRep a` where `a` is of a base kind $\star$ and returns a base element of that type. Thus, `baseElem TyBool` returns `False`, and `(TyApp TyList _)` maps to an empty list `[]`. System $\mathbf{F}_C$ is incapable of encoding the data constructors `TyList` and `TyApp` due to two reasons. First, the `TyRep` is kind polymorphic, requiring explicit kind abstraction and application. Second, the function implicitly uses kind equalities. In the case for `TyApp`, the first argument is inferred to have kind $\star \to \star$ and the second argument is inferred to have kind $\star$, making the result to be of kind $\star$.

9.1 Syntax

$$
\begin{array}{lrl}
 & \text{Type Vars} & \alpha, \beta, \gamma \\
 & \text{Term Vars} & x, y \qquad \text{Indices } \ i, n \in \mathbb{N} \\
 & \text{Coercion Vars} & c \\
\text{Type Constants} & T & ::= (\to) \mid \star \mid H \\
\text{Type level names} & w & ::= \alpha \mid F_n \mid T \\
\text{Propositions} & \zeta & ::= \tau \sim \sigma \\
\text{Types or coercions} & \phi & ::= \tau \mid \gamma \\
\text{Types and Kinds} & \tau, \sigma, \kappa & ::= w \mid \tau\ \tau \mid \forall \alpha{:}\kappa.\ \tau \mid \forall c{:}\zeta.\ \tau \mid \tau \blacktriangleright \gamma \mid \tau\ \gamma \\
\text{Coercions} & \nu, \gamma & ::= c \mid \langle \tau \rangle \mid \mathbf{sym}\ \gamma \mid \nu \circ \gamma \\
 & & \quad \mid \forall_{\nu}(\alpha_1, \alpha_2, c).\ \gamma \mid \nu\ (\gamma, \gamma') \mid \forall_{(\nu_1, \nu_2)}(c_1, c_2).\ \gamma \mid \gamma @ \nu \mid \gamma @ (\nu, \nu') \\
 & & \quad \mid \nu\ \gamma \mid \mathbf{left}\ \gamma \mid \mathbf{right}\ \gamma \mid \mathbf{nth}^i\ \gamma \mid \mathbf{kind}\ \gamma \mid T\ \overline{\phi} \\
\text{Patterns} & P & ::= H\ \overline{\alpha{:}\kappa}\ \Delta\ \overline{x{:}\tau} \\
\text{Telescopes} & \Delta & ::= \epsilon \mid \Delta, \alpha{:}\kappa \mid \Delta, c{:}\zeta \\
\text{Terms} & M, N & ::= x \mid \lambda x{:}\tau.\ M \mid M\ N \mid \Lambda \alpha{:}\kappa.\ M \mid M\ \tau \\
 & & \quad \mid \lambda c{:}\zeta.\ M \mid M\ \gamma \mid H \mid \mathsf{case}\ M\ \mathsf{of}\ \overline{P \to M} \mid M \blacktriangleright \gamma \\
\text{Typing Context} & \Gamma & ::= \epsilon \mid \Gamma, x{:}\tau \mid \Gamma, \alpha{:}\kappa \mid \Gamma, H{:}T \mid \Gamma, \gamma{:}\tau \sim \sigma
\end{array}
$$

Fig. 28. The Syntax of System $\mathbf{F}^{\kappa}_{C}$

The type of the data constructors in this system are of the form:

$$H : \forall \overline{\alpha{:}\kappa}.\ \forall\ \Delta\ .\ \overline{\tau} \to T\ \overline{\alpha}$$

The list $\overline{\alpha{:}\kappa}$ are the parameters of the data constructors. All the existential arguments to the data constructor are captured in the telescope Δ. The existential arguments can be types or coercions. In System $\mathbf{F}^{\kappa}_{C}$ there is no distinction between types and kinds and thus they can intermingle freely. The restriction on the language is that the data constructors can only be prameterized by types which are not coercions. For example, the type of `TyInt` and and `TyList` will be:

$$\texttt{TyInt} : \forall (k : \star)\ (t : k).\forall (coK : k \sim \star).\forall (co : t \sim Int).TyRep\ k\ t$$

$$\texttt{TyList} : \forall (k : \star)\ (t : k).\forall (coK : k \sim \star \to \star).\forall (co : t \sim [\,]).TyRep\ k\ t$$

In the above example, the `TyList` and `TyInt` is parameterized over the variables k and t while $\Delta = \{coK, co\}$. While the language of types and kinds is exactly the same, and there is no distinction between them, we will informally use the type as a classifier for terms and kinds as a classifier for types.

$$\boxed{\Gamma \vdash_c \gamma : \tau \sim \tau}$$

$$(\text{CO-EXT})\ \frac{\Gamma \vdash_c \gamma : \tau \sim \tau' \quad \Gamma \vdash_c \tau : \kappa \quad \Gamma \vdash_c \tau' : \kappa'}{\Gamma \vdash_c \mathbf{kind}\, \gamma : \kappa \sim \kappa'} \qquad (\text{CO-COH})\ \frac{\Gamma \vdash_c \gamma : \tau \sim \tau' \quad \Gamma \vdash \tau \blacktriangleright \gamma' : \kappa}{\Gamma \vdash_c \gamma \blacktriangleright \gamma' : \tau \blacktriangleright \gamma' \sim \tau'}$$

$$(\text{CO-}\forall\tau\ I)\ \frac{\begin{array}{c}\Gamma, \alpha{:}\kappa, \alpha'{:}\kappa', c{:}\alpha \sim \alpha' \vdash_c \gamma : \tau \sim \tau' \quad \Gamma \vdash_c \nu : \kappa \sim \kappa' \\ \Gamma \vdash \forall\alpha{:}\kappa.\ \tau : \star \qquad \Gamma \vdash \forall\alpha'{:}\kappa'.\ \tau' : \star\end{array}}{\Gamma \vdash_c \forall_\nu(\alpha, \alpha', c).\ \gamma : \forall\alpha{:}\kappa.\ \tau \sim \forall\alpha'{:}\kappa'.\ \tau'}$$

$$(\text{CO-}\forall\gamma\ I)\ \frac{\begin{array}{c}\Gamma, c{:}\zeta, c'{:}\zeta' \vdash_c \gamma : \tau \sim \tau' \qquad \{c, c'\}\#|\ \gamma\ | \qquad \Gamma \vdash_c \nu_1 : \sigma_1 \sim \sigma_1' \quad \zeta = \sigma_1 \sim \sigma_2 \\ \Gamma \vdash \forall c{:}\zeta.\ \tau : \star \quad \Gamma \vdash \forall c'{:}\zeta'.\ \tau' : \star \qquad\qquad \Gamma \vdash_c \nu_2 : \sigma_2 \sim \sigma_2' \quad \zeta' = \sigma_1' \sim \sigma_2'\end{array}}{\Gamma \vdash_c \forall_{(\nu_1,\nu_2)}(c, c').\ \gamma : \forall c{:}\zeta.\ \tau \sim \forall c'{:}\zeta'.\ \tau'}$$

$$(\text{CO-CAPP})\ \frac{\Gamma \vdash_c \gamma : \tau \sim \tau' \quad \Gamma \vdash \tau\, \nu : \kappa \quad \Gamma \vdash \tau'\, \nu' : \kappa'}{\Gamma \vdash_c \gamma\, (\nu, \nu') : \tau\, \nu \sim \tau'\, \nu'} \qquad (\text{CO-}\forall\gamma\ E)\ \frac{\Gamma \vdash_c \gamma : \forall c{:}\zeta.\ \tau \sim \forall c'{:}\zeta'.\ \tau' \quad \Gamma \vdash_c \nu : \zeta \quad \Gamma \vdash_c \nu' : \zeta'}{\Gamma \vdash_c \gamma @ (\nu, \nu') : \{c \mapsto \nu\}\tau \sim \{c' \mapsto \nu'\}\tau'}$$

$$\boxed{\Gamma \vdash_t \tau : \kappa}$$

$$(\text{TY-}\forall\gamma)\ \frac{\Gamma, c{:}\phi \vdash_t \tau : \star \quad \Gamma \vdash_p \phi\ \ \mathsf{ok}}{\Gamma \vdash_t \forall c{:}\phi.\ \tau : \star} \qquad (\text{TY-STAR})\ \frac{}{\Gamma \vdash_t \star : \star}$$

$$(\text{TY-CAST})\ \frac{\Gamma \vdash_t \tau : \kappa \quad \Gamma \vdash_c \gamma : \kappa \sim \kappa' \quad \Gamma \vdash_t \kappa' : \star}{\Gamma \vdash_t \tau \blacktriangleright \gamma : \kappa'}$$

$$\boxed{\Gamma \vdash_p \zeta\ \ \mathsf{ok}}$$

$$(\text{PROP-EQ})\ \frac{\Gamma \vdash \tau : \kappa \quad \Gamma \vdash \sigma : \kappa'}{\Gamma \vdash_p \tau \sim \sigma\ \ \mathsf{ok}}$$

Fig. 29. Excerpt of Static Semantics of System $\mathbf{F}_C^\kappa$

9.2 Static Semantics

Coercions are the proof terms which inhabit the propositional type equality kind $\tau \sim \sigma$. The **kind** construct makes the hetrogenous type equality possible by extracting (or building) the kind equality coercion from a type equality coercion by (CO-EXT). Similar to System $\mathbf{F}_C$ the type equality is an equivalence relation and it can also be decomposed using the $\mathbf{nth}^i\ \gamma$ rules. The system enforces proof irrelevance by not having any judgement for equality between two coercion proofs. This enables the system to trivially consider all coercions to be equivalent. The propositional equality check (PROP-EQ) ensures that the type equality is between two well-formed types. The coherence rule, (CO-COH), says that kind coercions can be ignored while proving type equality.

The rule (CO-$\forall$ *I*) in System $\mathbf{F}_C$ required that the kind of the quantified variable had to be syntactically equal. In System $\mathbf{F}_C^\kappa$, as seen in ($co - \forall\tau\ I$), that is no longer the case. It is possible that the variables have different kinds, but there is a coercion that makes them equal. The rule (CO-$\forall\gamma$ *I*) is analogus to ($co - \forall\tau\ I$) but for coercions. However, due to proof irrelevance, there is no proof witness required to show c and c' are equal.

9.3 Operational Semantics

Most of the operational semantics for System $\mathbf{F}_C^\kappa$ is carried over from System $\mathbf{F}_C$. The most interesting rule of the system is (S-KPUSH) which transforms a data constructor applied to a cast into an "equivalent" data constructor by pushing the cercions in the arguments. This is achieved by using the lifting operation.

$$\text{(S-KPush)}\ \frac{\begin{array}{c} H{:}\forall\overline{\alpha{:}\kappa}.\ \forall\Delta.\ \overline{\sigma} \to T\,\overline{\alpha} \quad \Psi = \texttt{extend}(\texttt{context}(\gamma); \overline{\phi}; \Delta) \\ \overline{\phi'} = \Psi_2(dom(\Delta)) \qquad \overline{M'} = \overline{M \blacktriangleright \Psi(\sigma)} \\ \overline{\tau'} = \Psi_2(\overline{\alpha}) \end{array}}{\texttt{case}\ (H\,\overline{\tau}\,\overline{\phi}\,\overline{M} \blacktriangleright \gamma)\ \texttt{of}\ \overline{\mathsf{P} \to N} \rightsquigarrow \texttt{case}\ H\,\overline{\tau'}\,\overline{\phi'}\,\overline{M'}\ \texttt{of}\ \overline{\mathsf{P} \to N}}$$

Fig. 30. Excerpt of Operational Semantics of System $\mathbf{F}^{\kappa}_{C}$

Definition 18 (Lifting operation, $\Psi(-)$). The lifting operation is defined inductively on the type structure:

$$\begin{aligned}
\Psi(\alpha) &= \gamma \text{ when } a{:}\kappa \mapsto (\tau_1, \tau_2, \gamma) \in \Psi \\
\Psi(\tau) &= \langle \tau \rangle \text{ when } \tau \# dom(\Psi) \\
\Psi(\tau\,\sigma) &= \Psi(\tau)\Psi(\sigma) \\
\Psi(\tau\,\gamma) &= \Psi(\tau)(\Psi_1(\gamma), \Psi_2(\gamma)) \\
\Psi(\forall\alpha{:}\kappa.\ \tau) &= \forall_{\Psi(\kappa)}(\alpha_1, \alpha_2, c).\Psi'(\tau) \\
&\qquad \text{where } \Psi' = \Psi, \alpha{:}\kappa \mapsto (a_1, a_2, c) \text{ and } \{\alpha_1, \alpha_2, c\} \# \Psi \\
\Psi(\forall\gamma{:}\sigma_1 \sim \sigma_2.\ \tau) &= \forall_{(\Psi(\sigma_1), \Psi\sigma_2)}(c_1, c_2).\Psi'(\tau) \\
&\qquad \text{where } \Psi' = \Psi, \alpha{:}\sigma_1 \sim \sigma_2 \mapsto (a_1, a_2, c) \text{ and } \{c_1, c_2\} \# \Psi \\
\Psi(\tau \blacktriangleright \gamma) &= \Psi(\tau) \blacktriangleright \mathbf{sym}\,((\mathbf{sym}\,\gamma) \blacktriangleright \Psi_1(\gamma)) \blacktriangleright \Psi_2(\gamma)
\end{aligned}$$

The lifting operation, $\Psi(-)$, is a generalized version of lifting defined to fomalize System $\mathbf{F}_C$ (Definition 1). The lifting context, Ψ, is built using the function $\texttt{context}(\gamma)$.

Definition 19 (Lifting Context Creation, $\texttt{context}(\gamma)$). If $\Gamma \vdash_c \gamma : T\,\overline{\tau} \sim T\,\overline{\sigma}$, and $T{:}\forall\overline{\alpha{:}\kappa}.\ {:}\star \in \Gamma$ then

$$\texttt{context}(\gamma) = \overline{\alpha_i{:}\kappa_i \mapsto (\tau_i, \sigma_i, \mathbf{nth}^i\,\gamma)}$$

Intuitively, $\texttt{context}(\gamma)_1(\tau)$ replaces all the occurances of α in τ with the corresponding "from" type and $\texttt{context}(\gamma)_2(\tau)$ replaces all the occurances of α in τ with the corresponding "to" type. The extend operation, $\texttt{extend}(\Psi; \overline{\phi}; \Delta)$ extends the initial context Ψ with the coercions present in the existential parameters of the data constructor telescope.

Definition 20 (Lifting Context Extension, $\texttt{extend}(\Psi; \overline{\phi}, \Delta)$).

$$\begin{aligned}
\texttt{extend}(\Psi; \epsilon; \epsilon) &= \Psi \\
\texttt{extend}(\Psi; \overline{\phi}, \tau; \Delta, \alpha{:}\kappa) &= \Psi', \alpha{:}\kappa \mapsto (\tau, \tau \blacktriangleright \Psi'(\kappa), \gamma') \\
&\qquad \text{where } \Psi' = \texttt{extend}(\Psi; \overline{\phi}; \Delta),\ \gamma' = \mathbf{sym}\,(\langle\tau\rangle \blacktriangleright \Psi'(\kappa)) \\
\texttt{extend}(\Psi; \overline{\phi}, \gamma; \Delta, c{:}\tau \sim \sigma) &= \Psi', c{:}\tau \sim \sigma \mapsto (\gamma, \gamma') \\
&\qquad \text{where } \Psi' = \texttt{extend}(\Psi; \overline{\phi}; \Delta),\ \gamma' = \mathbf{sym}\,(\Psi'(\tau)) \circ \gamma \circ \Psi'(\sigma)
\end{aligned}$$

As an illustration of how the lifting context aids the push rule consider the example:

```
Cons (T Bool) x xs ▶ γ
```

where, Cons:$\forall k{:}\star, a{:}k.\ \forall co{:}k \sim \star.\ a \to [a] \to [a]$, γ : `[T Bool] ~ [Int]`, and `T` is a type constant. The lifting context in this case:

$$\texttt{context}(\gamma) = \{a{:}\kappa \mapsto (\texttt{T Bool}, \texttt{Int}, \mathbf{nth}^1\ \gamma)\}$$

and the extension will be:

$$\Psi = \texttt{extend}(\texttt{context}(\gamma)) = \{a{:}\kappa \mapsto (\texttt{T Bool}, \texttt{Int}, \mathbf{nth}^1\ \gamma), co{:}\kappa \sim \star \mapsto (coK, \langle\star\rangle)\}$$

And thus,

$$\texttt{Cons k (T Bool) coK x xs} \blacktriangleright \gamma$$
$$\rightsquigarrow \texttt{Cons} \star\ \texttt{Int}\ (\langle\star\rangle)\ (\texttt{x} \blacktriangleright \Psi(a))\ (\texttt{xs} \blacktriangleright \Psi([a]))$$

Lemma 21 (Coercion lifting Lemma). *If Ψ is a valid lifting context for Γ and the telescope Δ, and $\Gamma, \Delta \vdash_t \tau : \kappa$ then*

$$\Gamma \vdash_c \Psi(\tau) : \Psi_1(\tau) \sim \Psi_2(\tau)$$

9.4 Meta-theory

With all the machinery built to allow kind hetrogenous equalities, it remains to argue whether the system still enjoys type safety.

Lemma 22 (Preservation for System $\mathbf{F}^{\kappa}_{C}$). *If* `Good` Γ, $\Gamma \vdash M : \tau$ *and* $M \rightsquigarrow M'$ *then* $\Gamma \vdash M' : \tau$

The proof is inspection of all the rules and making sure that the operational semantics rules do not change the type of the reduced term. The push rules are the most interesting rule to analyze. To ensure the push rules preserve types it is enough to check that erasure of type (an coercion) annotations from the term has no effect on the operational semantics.

Lemma 23 (Progress for System $\mathbf{F}^{\kappa}_{C}$). *If $\Gamma \vdash M : \tau$ and M is not CV, and Γ is a closed, consistent context, then there exists a M' such that $M \rightsquigarrow M'$*

The consistency criteria for the typing environment is captured using `Good` Γ and System $\mathbf{F}_C$. It is more involved as System $\mathbf{F}^{\kappa}_{C}$ is more complex than its predecessor systems. The proof proceeds in two steps: first, showing that the kind erasure does not interfere with semantics of types then second, the type erasure does not interfere with term semantics. There are some types in the system which are equal but cannot be proven so within the system. For example the two types $\forall c_1{:}Int \sim b.\ Int$ and $\forall c_2{:}Int \sim b.\ b$ are equal, but the system cannot prove them to be equal. Such incompleteness is a non-issue as the the surface level syntax disallows writing type equalities of the form $(Int \sim b) \Rightarrow Int \sim (Int \sim b) \Rightarrow b$ and these equalities are quite exotic.

System $\mathbf{F}^{\kappa}_{C}$ is a significantly more expressive type system than its predecessors. It achieves this by squashing the distinction between types and kinds. The judgement $\Gamma \vdash \star : \star$, or the type in type axiom, is one of the kinding axioms. It is usually responsible to creep inconsistencies in logics. However, System $\mathbf{F}^{\kappa}_{C}$ is no more inconsistent than System $\mathbf{F}_C$, where all kinds are inhabited. The typechecking problem for System $\mathbf{F}^{\kappa}_{C}$ is decidable and all the terms are explicitly type annotated. and the type equalities are always accompanied with a finitely sized equality proof. The system also maintains a distinction between the (equality) proofs and objects. This allows proofs to be independent of any computational value.

Part IV: The Path Ahead

10 Future Work

```
class Con e c | c ⇝ e where                  type family FDCon c : *
  empty :: ∀ e c. Con e c ⇒ c                empty : ∀ c. Con (FDCon c) c ⇒ c
  extend :: ∀ e c. Con e c ⇒ e → c → c       extend : ∀ c. Con (FDCon c) c ⇒ e → c → c

instance Con Int [Int] where                 axiom coFDConLIntInt : FDCon [Int] ~ Int
  empty = ...                                empty_[Int]  : Con [Int] (FDCon [Int]) ⇒ Int
  extend = ...                               extend_[Int] : Con [Int] (FDCon [Int]) ⇒ Int → [Int] → [Int]
```

Fig. 31. Elaborating Functional Dependencies into System $\mathbf{F}_C$

Functional dependencies introduce uniqueness criteria for a subset of the multiparameter type classes. We may expect obtaining a natural encoding of functional dependencies via the mechanism of open type functions. In this encoding scheme, each functional dependency induces a special type function that relates the determiner of the functional dependency to the determinant. Each typeclass instance introduces an axiom that encodes this mapping. Further, every type that mentions the determinant of the functional dependency gets replaced by the type function. For example, as shown in Figure 31, `Con e c` gets converted to `Con (FCCon c) c`. While this encoding is sound, and works for simple cases, it does not cover all the use cases. Consider a specialized case of containers, `UCon con`, for unsorted containers, where the container does not enforce any ordering on the elements.

```
class Con elem con ⇒ UCon con
```

Representing the typeclass constraints in a first order predicate logic, we get the following (seemingly broken) formula as the type variable `elem` is out of scope:

$$\forall con.\ \mathsf{UCon}\ con \Rightarrow \mathsf{Con}\ elem\ con$$

The intended meaning of the class declaration is: whenever we have an evidence that says the typeclass constraint `UCon con` is satisfied, we can also assume, or produce, an evidence for typeclass constraint `Con elem con`. Although it may seem that the type parameter `elem` is free in this definition, it is uniquely determined due to the functional dependency, `con ⇝ elem`. The type parameter `con` is existentially bound, as made explicit in the formula below:

$$\forall con.\ \mathsf{UCon}\ con \Rightarrow \exists! elem.\ \mathsf{Con}\ con\ elem$$

To tackle this problem, Karachalias and Schrijvers [2017] develop an elaboration scheme for typeclass and instance declarations which elaborates functional dependencies to type equality axioms in System $\mathbf{F}_C$. However, their work does not provide an explanation of the general theory of the simplifying and improvement of types [Jones, 1995], as functional dependencies may introduce partial type refinements. Functional dependencies remain a type inference level artifact and have no evidence at the level of System $\mathbf{F}_C$. The consequence of the incomplete encoding of functional dependencies is reflected in the compiler implementation. For example, issues with functional dependencies interacting

poorly with type families[1], GADTs[2], and furthermore, causing non-confluence issues in the GHC constraint solver [3]. These issues degrade the programmer's experience by obscure type errors, and worse the type checker is unnecessarily more restrictive than required.

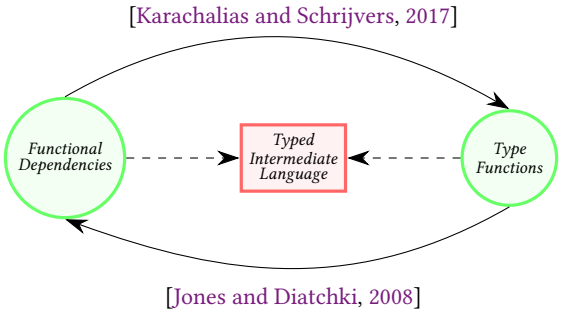

Fig. 32. Open Problem: Compiling Functional Dependencies and Type Functions to a typed Intermediate Language

An important research problem is to solve the tension between functional dependencies and type families by having a uniform treatment of both in a typed intermediate language as shown in Figure 32. In the next section we go over language extensions, some of which are implemented in GHC, while others are proposals to improve the programmer experience.

11 Related Work

11.1 Instance Chains

Instance chains [Morris and Jones, 2010] enable finer grained control of which instance should be used to satisfy a typeclass constraint. The instances for `Show` typeclass defines how a value can be converted to a string:

```
type String = [Char]
class Show a where show :: a → String
class Show a ⇒ Show [a] where show = ...
class Show String where show = ...
```

If we know how to write a `show` function for any value of a particular type, we can write a generic instance of show on a list of such types. However, we may want to override this behavior for some specific instances. For example, the type `String` in GHC is declared as a type synonym for a list of `Char`—(`type String = [Char]`). The typechecker now has two ways of resolving the typeclass constraint `Show String`: it can either use the generic instance via `Show [Char]`, or it can use the specific instance `Show String`. The typechecker has no way cannot decide which instance is more appropriate. Such instances are called *overlapping instances*. Haskell disallows overlapping instances as this gives rise to incoherence issues[Jones, 1993]. Instance chains solves this problem by allowing programmers to specify in what order the the instances can be resolved. It also subsumes closed typeclasses by providing a capability to express a default failure clause in an instance chain declaration.

[1]GHC Issue #11534: https://gitlab.haskell.org/ghc/ghc/-/issues/11534
[2]GHC Issue #345: https://gitlab.haskell.org/ghc/ghc/-/issues/345
[3]GHC issue #18851: https://gitlab.haskell.org/ghc/ghc/-/issues/18851

```
instance Show String where show s = ...
else instance Show a ⇒ [a] where show s = ...
else fail
```

11.2 Type Functions

11.2.1 Closed type families. The programmer may want to write type functions which has a restricted extension. For example, the previously defined `Add` type function is defined only on `Z` and `S n` types. Closed type families [Eisenberg et al., 2014] help in expressing such use cases. For example, consider the code snippet shown below:

```
type family Add m n where
  Add (S m) n = S (Add m n)
  Add _ n = n
```

An additional advantage is that the programmer can also define a default case handler (`Add _ n = n`) for such type function definitions. This declaration is inexpressible using open type functions as the axioms which are introduced by type function instances are order independent: the order in which the instances are declared is independent of the order in which they are searched by the solver. Writing closed type functions is very similar to writing case statements at term level. The difference is that instead of writing equations for terms, the user write equations for types. While this makes the system expressive and helps in making error messages related to type functions more user friendly, closed type families with non-linear type patterns—such as, `Add x x`—are reducible to term rewriting theories[Mizuhito, 1995], which are conjectured to guarantee confluence. The meta-theory needs to appeal to infinitary unification [Jaffar, 1984] of types to claim consistency. This is unsatisfactory, as Haskell does not have infinite types.

11.2.2 Injective type famililes. Type functions are non-injective by default. It can be a useful programming idiom to declare a type function to be injective so that the typechecker can use this information to rule out programs which violate the property. Further, the typechecker can also use this property to refine the type information. For example, consider an injective type function `F` declared to be injective below:

```
type family F a = r | r ⇝ a
type instance F Int = Char
type instance F Bool = Int
```

This syntax is a direct inspiration from functional dependencies for typeclasses. The declaration `r ⇝ a` says that the result of the type function `r`, can uniquely determine the argument of the type family `a`. Now, during type checking the instances of type functions, the typechecker would reject any axiom which would make `F` non-injective. For example, in the above code, the declaration `type instance F Char = Char` would be rejected by the typechecker. Further while trying to solve a wanted type equality constraint such as, `F a ~ Char`, the typechecker can leverage the fact that `F` is injective, and hence, there can be only one solution for the equation: `a ~ Int`, thus aiding the solver with extra information to solve the original wanted constraint. Injective type families [Stolarek et al., 2015] can be extended for closed type families as well. The key change in terms of formalization is to relax the rule (CO-NTH) from Figure 26 to allow decomposing injective type family applications. The formalization however, still has to assume that the type rewriting is terminating for proving consistency.

$$(\text{CO-NTH})\ \frac{\Gamma \vdash_c \gamma : \mathcal{T}\ \overline{\sigma} \sim_R \mathcal{T}\ \overline{\tau} \quad \mathcal{T} \in \{T, F\} (\text{F is injective})}{\Gamma \vdash_c \mathbf{nth}^i\ \gamma : \tau_1 \sim_{\rho_i} \tau_2}$$

11.2.3 Constrained Type Families. Constrained type families [Morris and Eisenberg, 2017] allows type computation to proceed only when the type family application can be provably reduced to a ground type. This simplifies the meta-theory of the language significantly. Constrained type families can prove confluence of type rewriting hence gives stronger type soundness guarantees. It avoids the pitfall of having to prove consistency by resorting to the use of infinitary unification and a conjecture to prove consistency as done in the meta theory of closed type families. Closed type classes, which otherwise would need special infrastructure, fall out naturally in this system. The formalized system however, does not have a corresponding working implementation[4].

12 Conclusion

Parametric Features	Non-parametric Features
Modules [MacQueen, 1984]	Typeclasses [Wadler and Blott, 1989] *Functional Dependencies* [Jones, 2000]
ADTs [Burstall et al., 1980]	*GADTs* [Cheney and Hinze, 2003]
Newtypes [Breitner et al., 2014]	*Open Type Functions* [Schrijvers et al., 2008] Closed Type Functions [Eisenberg et al., 2014], *Associated types* [Chakravarty et al., 2005]

Fig. 33. Features of Haskell

We surveyed some important language features–the ones which functional programmers prefer—and seen in depth, a theoretical foundation of a modern day declarative functional programming language. The language features are tabulated in the Figure 33. There are two ways of supporting a new programming language feature, either via encoding them with the help of libraries or, via supporting them as a core language feature. In the latter scenario, we realized that composing different language features is non-trivial and their subtle interactions can threaten semantic consistency. We first identified that the three seemingly different language features: newtypes, generalized algebraic datatypes and type functions can be encoded using a simple single construct: type equality. We then showed how non-parametric features interacting with parametric features of the language that contain explicit type equalities can cause type unsoundness bugs. Along the way we saw how the the type system evolved with different flavors of type equality: System $\mathbf{F}_C$ has a primitive notion defining when two types are equal, but System $\mathbf{F}_C^R$, and System $\mathbf{F}_C^\kappa$ make the type equality relation more refined. Taming ad-hoc language features requires sophisticated proof techniques to formalize and argue about their correctness. As new programming languages are designed, built and used, it is not only important to identify and carry over the good design principles established by theory using the formalization techniques, but also strengthen the foundational theory of programming languages by understanding the real requirements of practice.

References

A.V. Aho, R. Sethi, and J.D. Ullman. 2015. *Compilers, Principles, Techniques, and Tools.* Pearson India Education Services, India. https://www.worldcat.org/title/12285707

[4]GHC Proposal: https://github.com/typedrat/ghc-proposals/blob/constrained-type-families/proposals/0000-simple-constrained-type-families.rst

Arthur I. Baars and S. Doaitse Swierstra. 2002. Typing dynamic typing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming (ICFP '02)*. Association for Computing Machinery, New York, NY, USA, 157–166. doi:10.1145/581478.581494

Henk Barendregt, Wil Dekkers, and Richard Statman. 2013. *Lambda Calculus with Types*. Cambridge University Press, Cambridge. doi:10.1017/CBO9781139032636

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2014. Safe zero-cost coercions for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 189–202. doi:10.1145/2628136.2628141

R. M. Burstall. 1969. Proving Properties of Programs by Structural Induction. *Comput. J.* 12, 1 (1969), 41–48. doi:10.1093/comjnl/12.1.41

R. M. Burstall, D. B. MacQueen, and D. T. Sannella. 1980. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming* (1980-08-25) *(LFP '80)*. Association for Computing Machinery, New York, NY, USA, 136–143. doi:10.1145/800087.802799

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2005-01-12) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/1040305.1040306

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming* (2010-09-27) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/1863543.1863547

James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types*. Technical Report TR2003-1901. Cornell University. https://ecommons.cornell.edu/handle/1813/5614

Peter Dybjer. 1991. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, G. Plotkin and Gerard Huet (Eds.). Cambridge University Press, Cambridge, UK, 280–306. doi:10.1017/CBO9780511569807.012

Peter Dybjer. 1994. Inductive families. *Formal Aspects of Computing* 6, 4 (1994), 440–465. doi:10.1007/BF01211308

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, San Diego, California, USA, 671–683. doi:10.1145/2535838.2535856

Jacques Garrigue and Jacques Le Normand. 2011. *Adding GADTs to OCaml: the direct approach*. Technical Report. Nagoya University, Lexifi, Paris, France. https://www.math.nagoya-u.ac.jp/~garrigue/papers/ml2011.pdf

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

Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. 2020. Lower your guards: a compositional pattern-match coverage checker. *Proceedings of the ACM on Programming Languages* 4 (2020), 107:1–107:30. Issue ICFP. doi:10.1145/3408989

Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. 1994. Type classes in Haskell. In *Programming Languages and Systems — ESOP '94* (1994) *(Lecture Notes in Computer Science)*, Donald Sannella (Ed.). Springer, Berlin, Heidelberg, 241–256. doi:10.1007/3-540-57880-3_16

Sheon Han. 2023. *The Deep Link Equating Math Proofs and Computer Programs*. Quanta Magazine. https://www.quantamagazine.org/the-deep-link-equating-math-proofs-and-computer-programs-20231011/

Robert Harper, John C. Mitchell, and Eugenio Moggi. 1989. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989-12-01) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 341–354. doi:10.1145/96709.96744

Joxan Jaffar. 1984. Efficient unification over infinite terms. *New Generation Computing* 2, 3 (1984), 207–219. doi:10.1007/BF03037057

Mark P Jones. 1993. *Coherence for qualified types*. Research Report YALEU/DCS/RR-989. Yale University, New Haven, Connecticut. 10 pages. http://web.cecs.pdx.edu/~mpj/pubs/RR-989.pdf

Mark P. Jones. 1994. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, UK. doi:10.1017/CBO9780511663086

Mark P. Jones. 1995. Simplifying and Improving Qualified Types. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*. ACM, New York, NY, USA, 160–169. doi:10.1145/224164.224198

Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00)*. Springer-Verlag, Berlin, Germany, 230–244. doi:10.1007/3-540-46425-5_15

Mark P. Jones and Iavor S. Diatchki. 2008. Language and program design for functional dependencies. In *Proceedings of the first ACM SIGPLAN symposium on Haskell* (2008-09-25) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 87–98. doi:10.1145/1411286.1411298

Georgios Karachalias and Tom Schrijvers. 2017. Elaboration on functional dependencies: functional dependencies are dead, long live functional dependencies. *ACM SIGPLAN Notices* 52, 10 (2017), 133–147. doi:10.1145/3156695.3122966

J. L. Lassez, M. J. Maher, and K. Marriott. 1998. Unification revisited. In *Foundations of Logic and Functional Programming* (1988) *(Lecture Notes in Computer Science)*, Mauro Boscarol, Luigia Carlucci Aiello, and Giorgio Levi (Eds.). Springer, Berlin, Heidelberg, 67–113. doi:10.1007/3-540-19129-1_4

Xavier Leroy. 1995. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 142–153. doi:10.1145/199448.199476

David MacQueen. 1984. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming (LFP '84)*. Association for Computing Machinery, New York, NY, USA, 198–207. doi:10.1145/800055.802036

Simon Marlow and Simon Peyton Jones. 2010. *Haskell Language Report 2010*. The Haskell Foundation. https://www.haskell.org/onlinereport/haskell2010/

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. doi:10.1016/0022-0000(78)90014-4

R. Milner, Robin Milner, Mads Tofte, Harper Robert, and David MacQueen. 1997. *The Definition of Standard ML - Revised* (rev sub edition ed.). The MIT Press, Cambridge, MA, USA. Open Library ID: OL9529854M.

R. Milner, L. Morris, and M. Newey. 1975. A Logic for Computable Functions with Reflexive and Polymorphic Types. In *Proceedings of the Conference on Proving and Improving Programs*. IRIA-Laboria, France, 371–394.

Ogawa Mizuhito. 1995. *RTA open problem #79.* The RTA list of open problems. https://www.win.tue.nl/rtaloop/problems/79.html

Benoît Montagu and Didier Rémy. 2009-01-21. Modeling abstract types in modules with open existential types. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*. Association for Computing Machinery, New York, NY, USA, 354–365. doi:10.1145/1480881.1480926

J. Garrett Morris. 2014. A simple semantics for Haskell overloading. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell* (2014-09-03) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 107–118. doi:10.1145/2633357.2633364

J. Garrett Morris and Richard A. Eisenberg. 2017. Constrained Type Families. *Proc. ACM Program. Lang.* 1, ICFP, Article 42 (Aug. 2017), 28 pages. doi:10.1145/3110286

J. Garrett Morris and Mark P. Jones. 2010. Instance chains: type class programming without overlapping instances. *ACM SIGPLAN Notices* 45, 9 (2010), 375–386. doi:10.1145/1932681.1863596

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2011. Non-parametric parametricity. *Journal of Functional Programming* 21, 4 (2011), 497–562. doi:10.1017/S0956796811000165

Henrik Nilsson. 2005. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming (ICFP '05)*. Association for Computing Machinery, New York, NY, USA, 54–65. doi:10.1145/1086365.1086374

Bengt Nordström, Kent Petersson, and Jan M. Smith. 1990. *Programming in Martin-Lo f's type theory: an introduction.* Clarendon Press, USA.

Simon Peyton Jones and Shayan Najd. 2017. Trees that Grow. *The Journal of Universal Computer Science* 23, 1 (2017), 42–62. doi:10.3217/jucs-023-01-0042

Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. 2004. *Wobbly Types: Type Inference For Generalised Algebraic Data Types.* Technical Report MS-CIS-05-26. Microsoft Research. https://www.microsoft.com/en-us/research/publication/wobbly-types-type-inference-for-generalised-algebraic-data-types/

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press, Cambridge, Massachusetts.

Emil L. Post. 1947. Recursive Unsolvability of a problem of Thue. *The Journal of Symbolic Logic* 12, 1 (1947), 1–11. doi:10.2307/2267170

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium* (1974) *(Lecture Notes in Computer Science)*, B. Robinet (Ed.). Springer Berlin Heidelberg, Syracuse, NY, USA, 408–425.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type checking with open type functions. *ACM SIGPLAN Notices* 43, 9 (2008), 51–62. doi:10.1145/1411203.1411215

Tim Sheard and Emir Pasalic. 2008. Meta-programming With Built-in Type Equality. *Electronic Notes in Theoretical Computer Science* 199 (2008), 49–65. doi:10.1016/j.entcs.2007.11.012

Jeremy G. Siek. 2023. *Essentials of Compilation: An Incremental Approach in Racket.* The MIT Press, Cambridge, MA, USA.

Jan Stolarek, Simon Peyton Jones, and Richard A. Eisenberg. 2015. Injective type families for Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (2015-08-30) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 118–128. doi:10.1145/2804302.2804314

Christopher Strachey. 2000. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation* 13, 1 (2000), 11–49. doi:10.1023/A:1010000313106

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation* (2007-01-16) *(TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 53–66. doi:10.1145/1190315.1190324

The GHC Team. 2020. *The Glasgow Haskell Compiler.* The GHC Team. https://downloads.haskell.org/ghc/8.8.4/

TODO. 2050. TODO. In *Proceedings of TODO (TODO)*. TODO Corp, TODO Land, 0–0.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study.

Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84. doi:10.1145/2699407

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '89)*. ACM, Austin, Texas, USA, 60–76. doi:10.1145/75277.75283

Stephanie Weirich. 2000. Type-safe cast: (functional pearl). *ACM SIGPLAN Notices* 35, 9 (2000), 58–67. doi:10.1145/357766.351246

Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. 2011. Generative type abstraction and type-level computation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '11)*. Association for Computing Machinery, New York, NY, USA, 227–240. doi:10.1145/1926385.1926411

J. B. Wells. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98, 1 (1999), 111–156. doi:10.1016/S0168-0072(98)00047-5

A. K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. doi:10.1006/inco.1994.1093

Hongwei Xi. 1998. Dead Code Elimination through Dependent Types. In *Practical Aspects of Declarative Languages* (1998) *(Lecture Notes in Computer Science)*, Gopal Gupta (Ed.). Springer, Berlin, Heidelberg, 228–242. doi:10.1007/3-540-49201-1_16

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2003-01-15) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 224–235. doi:10.1145/604131.604150

Yichen Xu, Aleksander Boruch-Gruszecki, and Lionel Parreaux. 2021. Implementing path-dependent GADT reasoning for Scala 3. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala* (2021-10-17) *(SCALA 2021)*. Association for Computing Machinery, New York, NY, USA, 22–32. doi:10.1145/3486610.3486892

Christoph Zenger. 1997. Indexed types. *Theoretical Computer Science* 187, 1 (1997), 147–165. doi:10.1016/S0304-3975(97)00062-5