CIS 670 — Final project

Arthur Azevedo de Amorim
Bob Zhang

January 10, 2012

1 Introduction

With *generative type abstraction*, the implementation of a module can define a new type that is a synonym of some existing type while hiding this representation from clients of that module. Thus, whereas the implementation can treat the new type as an alias of the original type, code that uses that module is forced to use the exported interface, which might not even expose the actual representation of that type.

A problem arises when trying to combine this feature with other non-parametric features, such as type functions. Indeed, in some cases, it might be desirable for a type function to yield different results for two types even though those types are synonymous. Hence, in order to add these two features to a language in a sound manner, it is necessary to limit the extent to which two types can be synonymous.

In [**?**], the authors extend System FC, the core language of the Glasgow Haskell Compiler, with kind annotations that allow one to distinguish between parametric and non-parametric contexts in a program. Thus, when two types are declared synonymous, they can only be used as such in some contexts, while in others they are treated as being different.

In our work, we've implemented a slightly modified version of the FC_2 type checker. Section **??** discusses the main differences in the type system of FC_2 compared to that of System FC. Section **??** describes our implementation of the type checker.

2 Description

FC_2 adds two *kind roles* to base System FC, noted C (for "code") and T (for "type"). These mark two different kinds of equivalence and annotate type arguments in kinds. Equivalence at role C is a finer notion than equivalence at role T: types that are equivalent at role C are equivalent at role T, but the converse does not necessarily hold.

Intuitively, type functions that take arguments at role T care only about the representation of the types, but not about their names. Thus, if types `Int`

and `Age` are declared to be synonymous at role T, types `List Int` and `List Age` should be equivalent, even though they are symbolically different. This is the intended behavior for regular Haskell algebraic data types. However, if a type function takes arguments at role C, it'll distinguish types that have the same representation but different names. This case corresponds to indexed type families: it is possible to define a type family `F` such that `F Int = Char` but `F Age = Bool`. Thus, it would be an error to consider `F Int` and `F Age` as synonymous, even though the two arguments have the same representation.

As it stands, the FC_2 type system and context formation rules are too permissive to guarantee basic important properties, such as preservation and progress. Indeed, it is possible to add inconsistent type equivalence assumptions in the context, such as declaring that two distinct data types, such as `List a` and `Maybe a`, are equivalent. Hence, it is necessary to assume that a context is *consistent* in order to prove these safety theorems. The authors define context consistency and a sufficient property to show that a context is consistent. They also prove preservation and progress in these conditions for FC_2.

Finally, the authors show how to translate standard Haskell type declarations into a suitable FC_2 environment. The translation is such that appropriate roles are added to kinds in order to ensure that parametric and non-parametric contexts are treated correctly.

3 Implementation and Discussion

We've implemented a type checker for a version of FC_2 in OCaml. Our program follows essentially the typing rules shown in the paper. The rules are syntax-directed, so the translation was relatively straight-forward.

While working with the original version FC_2, we found that it would be worth adding some slight modifications to make the system more convenient to work with. The main difference lays in the way the context of a program is described. Instead of describing each part of the context separately, as done in the original system, our type declarations are modeled after Haskell's. Hence, a generic list data type is declared as

```
DATA List (a:*/T) WHERE {
  Cons :: {(a:*/T)} (a -> ((List a) -> (List a)))
  Nil :: {(a:*/T)} (List a)
};
```

while a Haskell-like newtype could be declared as

```
NEWTYPE Age = MkAge Nat;
```

Notice the explicit kind annotations: instead of inferring the best kinds and roles for each declaration, we decided to stick to the original system in most of the cases. Internally, the environment is still represented as a list of bindings. The notation is translated to the original form during parse time, inspired by

the compilation rules presented in the paper. Our parser was written using ocamllex and ocamlyacc.

We've also added the possibility of binding terms to names in the language using an environment for terms.

The type checker is divided into three components: a type checker for terms, which is the main function, a coercion proof checker and a kind checker for types. Implementing most of the cases was straight-forward: the most difficult one was typing a case statement, which requires manipulating the environment doing several verifications, but it was still fairly simple.

Due to time concerns, our implementation lacks some important features. It would be really useful to add some sanity checks to the environment, such as ensuring that it is well-formed or that the coercion assumptions are consistent, following the method presented in the paper. We also haven't implemented an evaluator for the system. Adding these features would be interesting directions for future work.

As a side note, while implementing, one of the main difficulties was trying to reason about the system while drawing our intuition from Haskell. Indeed, many features in Haskell, such as GADT's and existential types, are not directly present in System FC or FC_2 and need to be translated to those languages, sometimes in non-obvious ways.

4 Conclusion

We've implemented a type checker for FC_2, an extension of the core language of GHC that solves some problems related to type coercions in parametric and non-parametric contexts. The implementation is fairly basic, but captures the essence of its type system.