Solving the expression problem in Haskell with true separate compilation

Sean Seefried[1]
Manuel M. T. Chakravarty[2]

[1]Formal Methods
NICTA
Email: sean.seefried@nicta.com.au

[2]Programming Languages and Systems
School of Computer Science & Engineering
University of New South Wales
Email: chak@cse.unsw.edu.au

UNSW-CSE-TR-0715
June 2007

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

We present a novel solution to the expression problem which offers true separate compilation and can be used in existing Haskell compilers that support multi-parameter type classes and recursive dictionaries. The solution is best viewed as both a programming idiom, allowing a programmer to implement open data types and open functions, and the target encoding of a translation from Haskell augmented with syntactic sugar.

1. Introduction

The *expression problem* describes the difficulty of extending the variants *and* methods on a data type without modifying existing code and while respecting separate compilation. This problem has been well-studied and was first coined by Wadler [13] on the Java-Genericity mailing list. Although it originally described a specific problem—extending a program that processes terms of a simple programming language—it has come to represent the general problem of *extensible data types*. Zenger and Odersky [16] provide a good definition of the problem and a list of attendant criteria that a solution should satisfy. It is presented here with only minor paraphrasing.

- *Extensibility in both dimensions:* It should be possible to add new data variants and to introduce new functions.
- *Strong static type safety:* It should be impossible to apply a function to a data variant which it cannot handle.
- *No modification or duplication.* It should not be necessary to change existing code, nor should it be necessary to re-implement functionality when extending since this effectively amounts to duplication.
- *Separate compilation:* Compiling data type extensions or adding new functions should not encompass re-type-checking the original data type or existing functions, nor the re-compilation of existing modules. In this paper we aim for *true separate compilation* which involves just the compilation of new modules and must only require the interface files of existing modules.

A key observation made by Reynolds [11] and later echoed by others ([15], [5]) was that object-oriented and functional languages can be seen as complementary approaches to data abstraction. In object-oriented languages variants of a data type are modelled using *classes*; usually each variant is defined as a subclass of an abstract base class. Thus it is easy to add new variants. Unfortunately, the addition of new functionality on those variants is difficult; the only way to add new methods to a class is by sub-classing and it must be done for each variant. This quickly becomes unwieldy. In functional languages the converse is true: it is easy to add new functionality by defining new functions on a data type, but is difficult to add new variants. Another approach in object-oriented languages is to use the *visitor pattern* which makes it easy to add new functionality. However, as is the case with functional languages, adding new variants becomes difficult. Each of these approaches solves one half of the problem space but not the other.

A solution in Haskell has already been proposed by Löh and Hinze [9]. However, it differs from our solution in two ways. First, it does not provide true separate compilation for, at the very least, it is necessary to re-compile the *Main* module whenever an open declaration is added. Second, it relies on features that have not yet been implemented in any Haskell compiler. This is discussed further in Section 7.

The motivation in developing our solution was to provide extensibility for a compiler through plug-ins. Our compiler exposes data types—such as those representing the abstract syntax tree and type syntax—and functions that operated on those data structures. We wanted it to be possible for plug-in writers to extend both. This makes it possible, for instance, to write a plug-in syntactic sugar extension by adding new syntactic forms to the AST and new functions to desugar it to existing language constructs.

In a dynamic setting, such as a plug-in enabled application, a solution to the expression problem is absolutely necessary. Modifying source code is an intolerable option; one immediately loses the benefits of a plug-in compiler which include ease of extensibility and the ability to keep the source code a trade secret while allowing community participation in the development of its functionality. This also highlights why we require a solution that provides true separate compilation.

Our solution, while reliant on extensions to Haskell 98, works *as is*. It is presented as a translation from a simple syntactic extension to Haskell to existing Haskell syntax. However, the translation should be viewed from more than one angle. Naturally, the translation forms the basis for the implementation of a pre-processor. However, the target of the translation can also be seen as a *programming idiom* which can be readily used by developers to implement extensible data types *by hand*. It has already been used, in just such an idiomatic way, to implement front-end plug-ins for the aforementioned compiler.

The solution, henceforth known as *open abstract types*, uses several experimental features of Haskell: multi-parameter type classes, scoped type variables, kind annotations, zero constructor data types and recursive dictionaries. All of these features are present from GHC 6.4 onwards.

The structure of the rest of this paper is as follows: First, syntactic sugar is introduced for declaring extensible data types. Next, a running example is introduced, demonstrating the new syntax in action. At this point it is necessary to cover a (relatively complex) technique that is instrumental in the translation. In Section 4 the concept of *retrospective superclassing* is introduced. Without presenting the formalisation of the translation, Section 5, shows us the result of applying the translation and the most salient points of the code are discussed. Section 6 introduces the formal translation, which can be used as the basis for the implementation of a pre-processor. The paper concludes with a comparison of our solution to others in Haskell and a short discussion of solutions in other languages.

2. Syntactic sugar for open abstract types

Although the majority of this paper is concerned with demonstrating an encoding of extensible data type support in Haskell we are ultimately interested in introducing syntactic sugar to reduce its syntactic burden. In this section we present two new data declaration. In Section 6 an austere Haskell-like language augmented with these declarations becomes the source language in a formal translation to the encoding we are about to develop.

The two syntactic forms are *open data* and *extend data* declarations. A new extensible data type (EDT) is introduced with the **open data** keywords.

$$\textbf{module}\ F0\ \textbf{where}$$

$$\begin{aligned}\textbf{open data}\ \mathit{Exp}\quad &= \mathit{Var}\ \mathit{String}\\ &\mid \mathit{Lam}\ \mathit{String}\ \mathit{Exp}\\ &\mid \mathit{App}\ \mathit{Exp}\ \mathit{Exp}\end{aligned}$$

Functions can be defined upon these data types just like they can on ordinary algebraic data types.

$$\begin{aligned}&\mathit{alpha} :: \mathit{Exp} \to (\mathit{String}, \mathit{String}) \to \mathit{String}\\ &\mathit{alpha}\ (\mathit{Var}\ v) = \ldots\end{aligned}$$

In another module we can then extend the data type using the **extend data** keywords as follows:

$$\textbf{module}\ F1\ \textbf{where}$$

$$\textbf{extend data}\ \mathit{Exp} = \mathit{LetE}\ \mathit{String}\ \mathit{Exp}\ \mathit{Exp}$$

As usual it is possible to define new functions on the data type in this new module but in this case they can also be defined on the new Let variant.

$$\mathit{eval} :: \mathit{Exp} \to \mathit{Env} \to \mathit{Exp}$$

```
module F0_Alpha
where

open data Exp = Var String
    | Lam String Exp
    | App Exp Exp

alpha :: Exp → (String, String) → Exp
alpha (Var v         :: Exp) =
  λ(s :: (String, String)) → Var (swap s v)
alpha (Lam v body :: Exp) =
  λ(s :: (String, String)) →
     Lam (swap s v) (alpha body s)
alpha (App a b      :: Exp) =
  λ(s :: (String, String)) → App (alpha a s) (alpha b s)

swap :: (String, String) → String → String
swap ((a, b) :: (String, String)) =
  λ(o :: String) → if a == o then b else o
```

Figure 1. The initial module. It defines the data structure to represent the simple lambda calculus and an alpha conversion function.

```
eval (Var name) = ...
eval (Lam name body) = ...
eval (App f x) = ...
eval (LetE name body exp) = ...
```

Unlike regular Haskell, new equations for the functions defined in the first module can be defined. However, this can only be done for the new variants introduced. In this case we would be limited to a new equation on the *Let* variant.

```
alpha (Let name body exp) = ...
```

The semantics of pattern matching is slightly different than usual. Since new equations can be introduced on existing functions whenever an *extend data* declaration the meaning of the wild card pattern becomes ambiguous. Consider the situation where the wild card pattern is used both in module *F0* and *F1*. Which one should be used? Does the new one equation override the old one? In order to simply the presentation of this paper we have opted to disallow the wild-card pattern altogether. However, the *best-fit left-to-right* pattern matching solution devised by Löh and Hinze [9] could be implemented without too much trouble.

There are a few more restrictions on the new syntax. An *open data* and *extend data* declaration cannot appear in the same module. For a particular extensible data type there is at most one *extend data* declaration per module. It was stated earlier that new equations on existing functions *could* be defined. In fact, they *must* be; to omit them is an error.

3. A running example: the lambda calculus

As a running example we implement a data type representing the lambda calculus and two operations: alpha conversion and evaluation. At its simplest the lambda calculus consists of three core concepts: variables, abstraction and application.

We define two modules, an initial and one than extends the previous. The initial module appears in Figure 1 and defines the *alpha* function on a data type that represents just the core concepts of the lambda calculus.

We then extend the module in Figures 2 and 3. We add a new variant to the lambda calculus, *let expressions*. We then add a new equation for this variant to the *alpha* function and define two new functions, *eval* and *apply*.

```
module F1_Eval
where

import F0_Pretty
extend data Exp = Let String Exp Exp

alpha (LetE name body exp :: Exp) =
  λ(s :: (String, String)) →
     LetE (swap s name) (alpha body s) (alpha body s)
eval :: Exp → Env → Exp
eval (Var name :: Exp) =
  λ(env :: Env) → lookupEnv env name
eval (Lam name body :: Exp) =
  λ(env :: Env) → Lam name body
eval (App f x :: Exp)     =
  λ(env :: Env) → apply x env (eval f env)
eval (LetE name body exp :: Exp) =
  λ(env :: Env) → eval (App (Lam name exp) body) env
apply :: Exp → Env → Exp → Exp
apply (Var name :: Exp) =
  λ(env :: Env) (x :: Exp) → error "Function expected"
apply (Lam name body :: Exp) =
  λ(env :: Env) (x :: Exp) →
     eval body (extEnv env (name, eval x env))
apply (App f x :: Exp) =
  λ(env :: Env) (x :: Exp) → error "Function expected"
apply (LetE name body exp :: Exp) =
  λ(env :: Env) (x :: Exp) →
     error "Function expected"
```

Figure 2. The extension module. It extends the earlier data structure to represent let expression, defines an extra equation on the alpha conversion function and defines a new evaluation function.

```
type Env = [(String, Exp)]
lookupEnv :: Env → String → Exp
lookupEnv ([] :: Env) =
  λ(name :: String) →
     error $ "lookupEnv: Variable " ++
           show name ++ " not found"
lookupEnv (hd : tl :: Env) =
  λ(name' :: String) → lookupEnvAux hd tl name'
lookupEnvAux :: (String, Exp) → Env → String → Exp
lookupEnvAux ((name, term) :: (String, Exp)) =
  λ(rest :: Env) (name' :: String) →
     if name == name'
     then term else lookupEnv rest name'
extEnv :: Env → (String, Exp) → Env
extEnv = λ(env :: Env) (x :: (String, Exp)) → x : env
```

Figure 3. Some helper functions that are also present in the extension module.

The reader may notice that the functions are not defined as they usually would be. There is one at most one pattern match for each function and in each case the pattern match is flat (i.e. not nested). Also, the right-hand side of each function is a lambda expression which while legal Haskell is not standard idiom. In addition, readers may wonder why there is an $apply$ function at all when this could easily be defined as a case expression inside $eval$.

The translation presented later in this paper is complicated by many of the syntactically friendly features of Haskell such as where clauses, nested pattern matches, etc. To simplify the presentation the translation is assumed to be performed on an austere Haskell which includes the syntactic sugar introduced in Section 2. By presenting our running example in this austere Haskell it is hoped that the correspondence between the rules of the translation and the result of applying them to Figures 1, 2, and 3 is much more readily apparent.

4. Läufer's method and *retrospective superclassing*

In Section 5 a complete translation of the program in Figures 1 and 2 is presented. The solution is based on an extension to the work of Läufer [8] and involves a technique that we have dubbed *retrospective superclassing*. This section will outline Läufer's work, show a gap in the solution to the expression problem and present retrospective superclassing as a means of closing that gap. We also show why *recursive dictionaries*, a recent extension to Haskell, are necessary in order for retrospective superclassing to work.

In Haskell, type classes are the only candidate for *encoding* extensible data types since they are the only *open* declarations. Most declarations in Haskell are *closed*: their meaning is fully determined once and for all in the module they are written in. Their very nature precludes them from being used to encode extensible data types. However, *instance declarations*, which define the functionality of class methods for a given type, are open. They can be defined in a module that is not the same as the *class declaration* as long as they do not *overlap*[1] with an existing instance.

Läufer [8] introduced a technique similar to the dynamic dispatch mechanism of object-oriented languages which can be used as the basis for a solution to the expression problem. The key idea is to treat a class declaration as the *interface* to an abstract data type. Existential types are then used to "wrap" specific implementations of the abstract data type so that the only way to perform operations the data type is through class methods. These methods are available because the class context is "wrapped up" inside the existential type. The technique is demonstrated on our running example. Below, we introduce a class for the $alpha$ function and an existential type Exp wraps up differing value behind the $MkExp$ constructor. It shall be called the *wrapper type* from now on.

$$\begin{array}{l}\textbf{class}\ Alpha\ a\ \textbf{where}\\ \quad alpha\ ::\ a\ \rightarrow\ (String,\ String)\ \rightarrow\ Exp\end{array}$$

$$\textbf{data}\ Exp\ =\ forall\ a.\ Alpha\ a\ \Rightarrow\ MkExp\ a$$

Methods can then be defined on various data types but with the aid of an *unwrapping instance* can be applied to values of Exp and have the correct behaviour. The unwrapping instance provides us with a function of type $Exp\ \rightarrow\ (String,\ String)\ \rightarrow\ Exp$ as required. Its definition is quite simple.

$$\begin{array}{l}\textbf{instance}\ Alpha\ Exp\ \textbf{where}\\ \quad alpha\ (MkExp\ e)\ s\ =\ alpha\ e\ s\end{array}$$

[1] An overlap occurs when a given instance can be unified via substitution to another. e.g. $C(a, Int)$ overlaps with $C(Bool, b)$.

We now define *component types* and corresponding instances of the $Alpha$ class to represent the core lambda calculus and the let expression extension. The component types are called Exp_0 and Exp_1 respectively. Note that where we used to have recursive occurrences of the data type we now refer to the wrapper type.

$$\begin{array}{lcl}\textbf{data}\ Exp\_0 & = & Var\ String\\ & \mid & Lam\ String\ Exp\\ & \mid & App\ Exp\ Exp\end{array}$$

$$\textbf{instance}\ Alpha\ Exp\_0\ \textbf{where}\ \ldots$$

Exp_1 can be defined along with its instance in an entirely new module. Instances are open declarations.

$$\textbf{data}\ Exp\_1 = LetE\ String\ Exp\ Exp$$

$$\textbf{instance}\ Alpha\ Exp\_1\ \textbf{where}\ \ldots$$

4.1 The version problem

Let us now consider extending the functionality of the Exp data type by defining an interpreter on it. This will require a new class, $Eval$, to be defined. Using the inheritance mechanism of type classes we can require that $Alpha$ is a superclass of $Eval$.

$$\textbf{class}\ Alpha\ a\ \Rightarrow\ Eval\ a\ \textbf{where}\ \ldots$$

Unfortunately, this requires that we introduce a new type, say $EExp$, to wrap up this new class, since Exp only wraps up the $Alpha$ class.

$$\textbf{data}\ EExp\ =\ forall\ a.Eval\ a\ \Rightarrow\ MkEExp\ a$$

Without going any further we can see that there is going to be a problem. Once we have correctly defined instances on the component types and declared an unwrapping instance we will have a data type for which $eval$ and $alpha$ are both methods. However, while the type of $eval$ is $EExp\ \rightarrow\ Env\ \rightarrow\ EExp$ the type of $alpha$ is $EExp\ \rightarrow\ (String,\ String)\ \rightarrow\ Exp$. The return type is the original type. Unfortunately, this means the following expression would not type check: $eval\ (alpha\ (MkExp\ (Var\ "a"))\ ("a",\ "c"))\ []$.

4.2 Retrospective superclassing

Let us look more closely at why this problem occurs. When a value of type Exp is unwrapped the value extracted has access to all of class $Alpha$'s methods and those of its superclasses, *and no more*. At present there is no way that we can define the function $alpha$ to return values which will have access to methods that a programmer may write in the future.

The first hint of a solution becomes evident when we restate the methods a value of type Exp has access to, putting the emphasis in a different place this time: it has access to all of $Alpha$'s methods *and those of its superclasses*, and no more. If it were somehow possible to define $Eval$ in such a way that it was a superclass of $Alpha$ then values of type Exp would have access to these methods. This would be a kind of *retrospective superclassing*.

In fact, retrospective superclassing is possible using a technique due to Hughes [3] and elaborated upon by Lämmel and Peyton Jones [7] which allows abstraction over type classes. Hughes' suggestion was to allow declarations like the following:

$$\begin{array}{l}\textbf{class}\ cxt\ a\ \Rightarrow\ Alpha\ cxt\ a\ \textbf{where}\\ \quad alpha\ ::\ a\ \rightarrow\ (String,\ String)\ \rightarrow\ Exp\ cxt\end{array}$$

$$\textbf{data}\ Exp\ cxt\ =\ forall\ a.\ Alpha\ cxt\ a\ \Rightarrow\ MkExp\ a$$

This is not valid Haskell since the second parameter, cxt, of the $Alpha$ class stands for a *class*, not a type or type constructor.

However, let us assume for the moment that such declarations are legal. Now type Exp has an extra parameter, cxt, which abstracts over a class. Since this very same class is declared to be a superclass of $Alpha$ we see that method $alpha$ now returns values which have access to the methods in any class that cxt is instantiated to.

Fortunately, Hughes was successful in encoding just such an abstraction over classes and the technique is now demonstrated. First, we define a class Sat with a single method $dict$. This class is used to return an *explicit dictionary* whose values are taken directly from the implicit one associated with a given class.

```
class Sat a where
  dict :: a
```

Now, whenever the programmer defines a new class they also define a corresponding data type that represents explicitly the implicit dictionary of the class. The programmer also needs to define an instance that equates the methods of the explicit dictionary with those classes we wish to abstract over. The following self-contained example demonstrates this.

```
type Env = [(String, Exp EvalD)]
class Sat (cxt a) ⇒ Alpha cxt a where
  alpha :: a → (String, String) → Exp cxt
class Alpha EvalD a ⇒ Eval a where
  eval :: a → Env → Exp EvalD
data EvalD a = EvalD{ eval′ :: a → Env → Exp EvalD}
instance Eval a ⇒ Sat (EvalD a) where
  dict = EvalD{ eval′ = eval}
```

Here is a quick summary of the salient points:

- The class head, $class\ cxt\ a \Rightarrow Alpha\ a$, has become $class\ Sat\ (cxt\ a) \Rightarrow Alpha\ a$.
- $EvalD$ is the explicit analogue of the implicit dictionary that is associated with the $Eval$ class.
- The instance equates the methods of $Eval$ with the explicit dictionary $EvalD$.

There is one remaining caveat – calls to extension methods must be done through explicit dictionaries. The following expression will not type check since method $eval$ is not a member of any superclass of $Alpha$.

```
case alpha exp ("a", "b") of MkExp exp′ → eval exp′ []
```

However, $dict$ is a method of $Alpha$'s superclass, Sat. All that is required is to replace $eval\ exp'\ []$ with $eval'\ dict\ exp'[]$ which only imposes minor syntactic inconvenience.

Retrospective superclassing relies on *recursive dictionaries*, a recently[2] implemented feature of GHC. These dictionaries allow cycles to occur while resolving the constraints introduced by class and instance declarations. We defer an in depth discussion of this to Section 5.3 but refer the reader to Lämmel and Peyton Jones' paper [7] on extensible generic functions where the technique was first described.

In conjunction with capping classes, the explicit dictionary of the Sat instance "ties the knot" of constraint resolution. This brings the functionality introduced by each class—in this case $Alpha$ and $Eval$— to the same semantic level. In Section 6.4.2 we will see that it is possible to call extension functions from new equations on existing functions.

[2] Recursive dictionaries are available from GHC 6.4 onwards.

```
module F0_Alpha
where

data P d
class Sat a where
  dict :: a
data Exp (cxt :: ∗ → ∗) =
  forall b. Alpha cxt b ⇒ MkExp b
```

Figure 4a. Preliminaries: the Sat class and wrapper type

```
class Sat (cxt b) ⇒ Alpha cxt b where
  alpha :: P cxt → b → (String, String) → Exp cxt
data Exp_0 cxt = Var String
               | Lam String (Exp cxt)
               | App (Exp cxt) (Exp cxt)
```

Figure 4b. Initial component type and the initial functionality class

```
instance (Sat (cxt (Exp cxt))
         , Sat (cxt (Exp_0 cxt))
         ) ⇒ Alpha cxt (Exp_0 cxt) where
  alpha (_ :: P cxt) (Var v :: Exp_0 cxt) =
    λ(s :: (String, String)) → var (u :: P cxt) (swap s v)
  alpha (_ :: P cxt) (Lam v body :: Exp_0 cxt) =
    λ(s :: (String, String)) →
      lam (u :: P cxt) (swap s v)
                       (alpha (u :: P cxt) body s)
  alpha (_ :: P cxt) (App a b :: Exp_0 cxt) =
    λ(s :: (String, String)) →
      app (u :: P cxt)
        (alpha (u :: P cxt) a s) (alpha (u :: P cxt) b s)
```

Figure 4c. Functionality instance

```
instance Sat (cxt (Exp cxt))
           ⇒ Alpha cxt (Exp cxt) where
  alpha (_ :: P cxt) (MkExp e :: Exp cxt) =
    λ(s :: (String, String)) → alpha (u :: P cxt) e s
```

Figure 4d. Unwrapping instance

5. Translation of the running example

We are now ready to discuss the translation of the initial module (Figure 1) and the extension module (Figures 2 and 3) of section 3.

To avoid overwhelming the reader the translation has been broken up into several sub-figures. The translation of the initial module appears in Figures 4a through 4g and the translated extension module in Figures 5a through 5h.

5.1 Initial module

Figure 4a introduces the Sat class and the wrapper type which, this time, contains a kind annotation. Although not strictly necessary in this case, it is required when the open data type has type parameters. We also introduce a *proxy type*, P. An argument of the proxy type is

```
data AlphaEnd b
class Alpha AlphaEnd b ⇒ AlphaCap b
instance AlphaCap (Exp_0 AlphaEnd)
instance AlphaCap (Exp AlphaEnd)
instance AlphaCap b ⇒ Sat (AlphaEnd b) where
  dict = error "Capped at Alpha"
```

Figure 4e. Capping classes, capping types and capping instances

```
var :: forall cxt. (Sat (cxt (Exp cxt))
                   , Sat (cxt (Exp_0 cxt))) ⇒
         P cxt → String → Exp cxt
var (_ :: P cxt) =
  λ(x1 :: String) → MkExp (Var x1 :: Exp_0 cxt)
lam :: forall cxt. (Sat (cxt (Exp cxt))
                   , Sat (cxt (Exp_0 cxt))) ⇒
         P cxt → String → Exp cxt → Exp cxt
lam (_ :: P cxt) = λ(x1 :: String) (x2 :: Exp cxt) →
                     MkExp (Lam x1 x2 :: Exp_0 cxt)
app :: forall cxt. (Sat (cxt (Exp cxt))
                   , Sat (cxt (Exp_0 cxt))) ⇒
         P cxt → Exp cxt → Exp cxt → Exp cxt
app (_ :: P cxt) = λ(x1 :: Exp cxt) (x2 :: Exp cxt) →
                     MkExp (App x1 x2 :: Exp_0 cxt)
```

Figure 4f. Smart constructors

```
swap :: (String, String) → String → String
swap ((a, b) :: (String, String)) =
  λ(o :: String) → if a == o then b else o
```

Figure 4g. Regular declarations

required[3] whenever the type signature of a method does not contain an occurrence of the cxt parameter. It is required for the correct unification of types. This is described in Section 6.4.1.

Figure 4b defines the initial functionality class, $Alpha$ and the initial component type Exp_0. The functionality instance of Figure 4c defines the three equations of the $alpha$ method on the Var, Lam and App variants of type Exp_0. There are two important things to note. First, there are two Sat constraints in the instance head, one on the initial component type and one on the wrapper type. The one for the wrapper type is necessary since $alpha$ returns a value of type $Exp\ cxt$. Second, use is made of the smart constructors var, lam and app defined in Figure 4f. These simplify the presentation considerably and are also useful when constructing concrete values of type $Exp\ \tau$ (for some type τ).

We call the $swap$ function in Figure 4g a *regular declaration* since it is not defined directly upon the open data type. Although it is unchanged in this translation this will not always be the case. Should a function use one of the instance methods its type will need to be augmented. More is said about this in Section 6.

The only remaining figure to explain is Figure 4e. A *capping class* is a null extension that allows a programmer to use the EDT in its current state. A capping class is always accompanied by a Sat instance featuring the capping class as its superclass. (In this case the capping class is $AlphaCap$.)

[3] The proxy type is not strictly required for this example either.

```
module F1_Eval
where
import F0_Alpha
data Exp_1 (cxt :: * → *) =
  LetE String (Exp cxt) (Exp cxt)
```

Figure 5a. Module header and new component type

```
instance (Sat (EvalD cxt (Exp (EvalD cxt)))
         , Sat (EvalD cxt (Exp_0 (EvalD cxt)))
         , Sat (EvalD cxt (Exp_1 (EvalD cxt)))
         ) ⇒ Alpha (EvalD cxt) (Exp_1 (EvalD cxt))
where
  alpha (_ :: P (EvalD cxt))
        (LetE name body exp :: Exp_1 (EvalD cxt)) =
    λ(s :: (String, String)) →
      letE (u :: P (EvalD cxt)) (swap s name)
           (alpha (u :: P (EvalD cxt)) body s)
           (alpha (u :: P (EvalD cxt)) exp s)
```

Figure 5b. Instances for new equations on existing functions

```
class (Sat (EvalD cxt b)
      , Alpha (EvalD cxt) b) ⇒ Eval cxt b where
  eval   :: P (EvalD cxt) → b → Env (EvalD cxt) →
              Exp (EvalD cxt)
  apply  :: P (EvalD cxt) → b → Env (EvalD cxt) →
              Exp (EvalD cxt) → Exp (EvalD cxt)
data EvalD cxt b =
  EvalD{ eval'   :: P (EvalD cxt) → b →
                      Env (EvalD cxt) →
                      Exp (EvalD cxt)
       , apply'  :: P (EvalD cxt) → b →
                      Env (EvalD cxt) →
                      Exp (EvalD cxt) →
                      Exp (EvalD cxt)
       , evalExt :: cxt b }
```

Figure 5c. Functionality classes and explicit dictionary

5.2 Extension module

The first thing to notice about Figures 5a through 5h is that the type variable cxt has been replaced almost wholesale by $EvalD\ cxt$. $EvalD$ is the name of the explicit dictionary defined in Figure 5c and its occurrence in the type $Exp\ (EvalD\ cxt)$ gives a visual indication that evaluation is defined upon it. Although we present no more functionality for the Exp EDT it is readily extensible. As more functionality is added the cxt type variable is replaced with further explicit dictionaries, e.g. $Exp\ (EvalD\ (Pretty\ cxt))$ and so on.

The *extension functionality class* is shown in Figure 5c. In general there will be one of these present in the translation whenever a new function is defined on the EDT.

Figure 5e, while much larger than the corresponding code in Figure 2 is a relatively straightforward translation of what is present

```
instance Sat (EvalD cxt (Exp (EvalD cxt))) ⇒
        Eval cxt (Exp (EvalD cxt)) where
  eval (_ :: P (EvalD cxt))
       (MkExp e :: Exp (EvalD cxt)) =
    λ(x1 :: Env (EvalD cxt)) →
      eval′ dict (u :: P (EvalD cxt)) e x1
  apply (_ :: P (EvalD cxt))
        (MkExp e :: Exp (EvalD cxt)) =
    λ(x1 :: Env (EvalD cxt)) (x2 :: Exp (EvalD cxt)) →
      apply′ dict (u :: P (EvalD cxt)) e x1 x2
```

Figure 5d. Unwrapping instance

there. One key difference is that uses of $eval$ and $apply$ on the right hand sides of the equations have been replaced with calls to $eval'\ dict$ and $apply'\ dict$ respectively. This occurs in any instances of extension functionality classes.

Figure 5f introduces the capping classes, types and instances. Note that this time the methods of class $Eval$, $eval$ and $apply$ are equated with the selector methods of $EvalD$, $eval'$ and $apply'$. The selector method $evalExt$ is equated with an error, much like $dict$ was in Figure 4e. As more functionality is added to the Exp EDT the $dict$ method of the Sat instance will come to consist of nested explicit dictionaries. Figure 8c provides more detail.

The *regular declarations* of Figure 5h have changed in the translation. The Env type now has a cxt parameter because it references the Exp type. Similarly the types of $lookupEnv$, $lookupEnvAux$ and $extEnv$ have changed.

5.3 Recursive dictionaries

In conjunction with *capping instances* the “knot” of class constraint dependency is “tied” via the Sat instance. Also, the capping type—in this case $AlphaEnd$—allows concrete values of the EDT to be created.

A recursive dictionary is created for (and only for) each instance of the capping class. Figure 6 graphically represents the structure of the two recursive dictionaries created for the Exp_0 and Exp types. (Interestingly, one of the dictionaries contains the other.) To see how they are built consider what happens when type checking $instance\ AlphaCap\ (Exp\ AlphaEnd)$. First, we must check if an instance of the superclass exists. The leads to the following constraint chain.

$$\begin{array}{l} \quad Alpha\ AlphaEnd\ (Exp\ AlphaEnd) \\ \leadsto\ Sat\ (AlphaEnd\ (Exp\ AlphaEnd)) \\ \leadsto\ AlphaCap\ (Exp\ AlphaEnd) \end{array}$$

We are back where we started. Fortunately, recursive dictionaries allow such cyclic constraints to be resolved. A similar line of reasoning shows us how the $instance\ AlphaCap\ (Exp\_0\ AlphaEnd)$ is typed and it is graphically represented in Figure 6. The boxes outlined by broken lines represent dictionary transformers (which correspond to instances with contexts). One can also read the solid arrows as *application* to the box at its tip. Following Wadler and Blott’s [14] original formulation of dictionary translation we can see the form of the recursive dictionary in d.

$$\begin{array}{l} d\ ::\ AlphaCapD\ (Exp\ AlphaEnd) \\ d\ =\ AlphaCapD\ \{\ alphaD\ =\ dt_1\ (dt_2\ d\ \} \\ dt_1\ ::\ SatD\ cxt\ (Exp\ cxt)\ \rightarrow\ AlphaD\ cxt\ (Exp\ cxt) \\ dt_1\ =\ \ldots \\ dt_2\ ::\ AlphaCapD\ b\ \rightarrow\ SatD\ AlphaEnd\ b \\ dt_2\ =\ \ldots \end{array}$$

```
instance (Sat (EvalD cxt (Exp (EvalD cxt)))
         , Sat (EvalD cxt (Exp_0 (EvalD cxt)))
         , Sat (EvalD cxt (Exp_1 (EvalD cxt)))
         ) ⇒ Eval cxt (Exp_0 (EvalD cxt)) where
  eval (_ :: P (EvalD cxt))
       (Var name :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) → lookupEnv env name
  eval (_ :: P (EvalD cxt))
       (Lam name body :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) →
      lam (u :: P (EvalD cxt)) name body
  eval (_ :: P (EvalD cxt))
       (App f x :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) →
      apply′ dict (u :: P (EvalD cxt)) x env
          (eval′ dict (u :: P (EvalD cxt)) f env)
  apply (_ :: P (EvalD cxt))
        (Var v :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt))
      (x :: Exp (EvalD cxt)) →
      error "Function expected"
  apply (_ :: P (EvalD cxt))
        (Lam name body :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt))
      (x :: Exp (EvalD cxt)) →
      eval′  dict (u :: P (EvalD cxt)) body
        (extEnv env (name,
                     eval′ dict (u :: P (EvalD cxt))
                         x env))
  apply (_ :: P (EvalD cxt))
        (App f x :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt))
      (x :: Exp (EvalD cxt)) →
      error "Function expected"
instance (Sat (EvalD cxt (Exp (EvalD cxt)))
         , Sat (EvalD cxt (Exp_0 (EvalD cxt)))
         , Sat (EvalD cxt (Exp_1 (EvalD cxt)))
         ) ⇒ Eval cxt (Exp_1 (EvalD cxt)) where
  eval (_ :: P (EvalD cxt))
       (LetE name body exp :: Exp_1 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) →
      eval′  dict (u :: P (EvalD cxt))
          (app (u :: P (EvalD cxt))
               (lam (u :: P (EvalD cxt)) name exp)
               body) env
  apply (_ :: P (EvalD cxt))
        (LetE name body exp :: Exp_1 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt))
      (x :: Exp (EvalD cxt)) →
      error "Function expected"
```

Figure 5e. Instances for new functions on all component types

```
data EvalEnd b
class Eval EvalEnd b ⇒ EvalCap b
instance EvalCap (Exp (EvalD EvalEnd))
instance EvalCap (Exp_0 (EvalD EvalEnd))
instance EvalCap (Exp_1 (EvalD EvalEnd))
instance EvalCap b ⇒ Sat (EvalD EvalEnd b) where
  dict = EvalD{ eval'  = eval
              , apply' = apply
              , evalExt = error "Capped at Eval"}
```

Figure 5f. Capping class, capping type and capping instances

```
letE :: forall cxt.
        (Sat (EvalD cxt (Exp (EvalD cxt)))
        , Sat (EvalD cxt (Exp_0 (EvalD cxt)))
        , Sat (EvalD cxt (Exp_1 (EvalD cxt)))) ⇒
        P (EvalD cxt) → String → Exp (EvalD cxt) →
        Exp (EvalD cxt) → Exp (EvalD cxt)
letE (_ :: P (EvalD cxt)) =
  λ(x1 :: String) (x2 :: Exp (EvalD cxt))
   (x3 :: Exp (EvalD cxt)) →
   MkExp (LetE x1 x2 x3 :: Exp_1 (EvalD cxt))
```

Figure 5g. Smart constructors

```
type Env cxt = [(String, Exp cxt)]
lookupEnv :: Env (EvalD cxt) → String →
             Exp (EvalD cxt)
lookupEnv ([] :: Env (EvalD cxt)) =
  λ(name :: String) →
    error ("lookupEnv : Variable " ++
           show name ++ " not found")
lookupEnv (hd : tl :: Env (EvalD cxt)) =
  λ(name' :: String) → lookupEnvAux hd tl name'
lookupEnvAux :: (String, Exp (EvalD cxt)) →
  Env (EvalD cxt) → String →
  Exp (EvalD cxt)
lookupEnvAux (name, term) =
  λ(rest :: Env (EvalD cxt)) (name' :: String) →
    if name == name'
    then term else lookupEnv rest name'
extEnv :: Env (EvalD cxt) →
          (String, Exp (EvalD cxt)) → Env (EvalD cxt)
extEnv = λ(env :: Env (EvalD cxt))
           (x :: (String, Exp (EvalD cxt))) → x : env
```

Figure 5h. Regular declarations

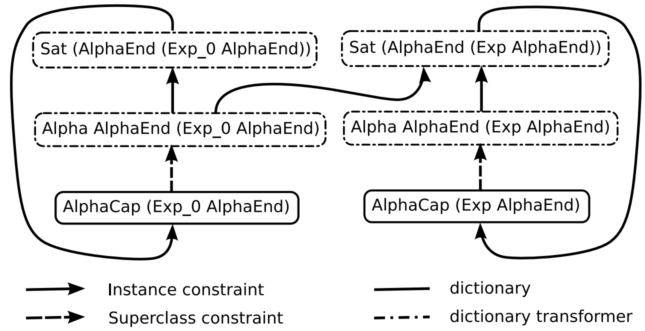

Figure 6. A diagram of two recursive dictionaries produced by *AlphaCap* instances on *Exp* and *Exp_0*.

Symbol Classes

α, β, γ	$\rightarrow$	$\langle$type variable$\rangle$	
T, E	$\rightarrow$	$\langle$type constructor$\rangle$	
$\mathcal{C}, \mathcal{E}$	$\rightarrow$	$\langle$data constructor$\rangle$	
x, f	$\rightarrow$	$\langle$term variable$\rangle$	
$\overline{\nu}$	$\rightarrow$	$\langle$Collection of pattern variables$\rangle$	
Declarations			
pgm	$\rightarrow$	$\overline{decl}$	(whole program)
$decl$	$\rightarrow$	$data;\ tval$	(declaration)
$data$	$\rightarrow$	$\mathbf{data}\ T\ \overline{\alpha}\ =\ \mathcal{C}\ \overline{\tau}$	(data type decl)
val	$\rightarrow$	$x\ =\ e \mid x\ p\ =\ e$	(value binding)
$vsig$	$\rightarrow$	$x : \sigma$	(type signature)
$tval$	$\rightarrow$	$vsig\ ;\ \overline{val}$	(top level binding)
Terms (Expressions)			
$e,\ b$	$\rightarrow$	$e_1\ e_2 \mid \lambda x : \tau . e \mid x \mid \mathcal{C}$	
Patterns			
p	$\rightarrow$	$\mathcal{C}\ x_1\ \ldots\ x_n\ :\ \tau$	$(n \geq 0)$(pattern)
Types			
τ, ξ	$\rightarrow$	$T \mid \alpha \mid \tau_1\ \tau_2$	(monotype)
σ	$\rightarrow$	$\tau \mid \forall \alpha . \sigma$	(type scheme)

Figure 7a. Syntax of source language

A full dictionary translation of the code in Figures 4a – 4g appears in Appendix B.

6. Formalisation

In this section we present a formal translation from the language described in Section 2 to Haskell. However, so that we may concentrate on the important aspects we translate from an austere source language to a target language equivalent in expressiveness to Haskell. The running example, although legal Haskell, was written in a manner very close to the source language which is essentially the lambda calculus with algebraic data types, flat pattern matching and first order polymorphic types. Most importantly, it contains two new forms of algebraic data type declarations: *open data* and *extend data*.

The target language has type classes but the syntactic restrictions on them are less stringent than Haskell 98. The source language does not contain type classes but only in order to simplify the presentation.

6.1 The source and target languages

Apart from the *open data* and *extend data* declarations the lexical structure of the source language does not differ much from the lambda calculus extended with algebraic data types and pattern matching. However there are a number of non-lexical restrictions on the syntax. These have largely been put in place to simplify the presentation of the translation and, in such cases, other translations from the richer language constructs of full Haskell are known to exist. Some constraints are essential but these have already been enumerated in Section 2. This section will only describe those constraints that simplify the presentation.

There is at most one pattern match per function and it must be flat, i.e. not nested. The source language is explicitly typed. All functions have type signatures except new equations on existing functions. This is because signatures already exist for such equations albeit in a different module. It is an error to provide signatures for them.

Further, all value bindings in the source language are supercombinators. We overload the terminology and allow both value bindings and expressions to be supercombinators. An expression that is supercombinator has the form:

$\lambda x_1 : \tau_1 \ldots \lambda x_n : \tau_n . e$

It has the following properties.

- It has no free variables.
- Any sub-term in e that is a lambda abstraction is also a supercombinator.
- $n \geq 0$.

A value binding that is a supercombinator has the form:

$x\ p\ =\ \lambda x_1 : \tau_1 \ldots \lambda x_n : \tau_n . e$

- It has no free variables.
- The pattern, p, is optional if the function is not defined on an EDT. Otherwise it is required.
- Any sub-term in e that is a lambda abstraction is a supercombinator.

This restriction was introduced so that it would not be necessary to deal with *let expressions* and *where* clauses. Using lambda-lifting it is always possible to translate from a language containing these to one of supercombinators.

6.1.1 Syntactic conventions

The syntax is provided in Figure 7a. Overbar notation is used extensively. The notation $\overline{\alpha}^n$ means the sequence $\alpha_1 \ldots \alpha_n$; the "n" may be omitted when it is unimportant. The following notational shortcuts also apply:

$$\begin{aligned} \overline{\tau}^n \rightarrow \xi &\equiv \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \xi \\ \forall \overline{\alpha}^n . \tau &\equiv \forall \alpha_1 \ldots \forall \alpha_n . \tau \end{aligned}$$

Superscripts and subscripts make a difference to what overbars mean. $\overline{D}_i^m\ \delta$ $(1 \leq i \leq m)$ is shorthand for $D_i\ (D_{i+1}\ \ldots\ (D_m\ \delta) \ldots)$. $\overline{D}^m\ \delta$ is shorthand for $\overline{D}_1^m\ \delta$. $\overline{D}_i^m\ \delta$ is the type of an explicit dictionary for functionality class F_i with the explicit dictionaries for functionality classes $F_{i+1}, \ldots, F_m$ nested within it. Also, we accommodate function types $\tau_1 \rightarrow \tau_2$ by regarding them as the curried application of the function type constructor to two arguments, thus: $(\rightarrow)\tau_1\tau_2$.

The following conventions apply to the symbols used. The first symbol appearing in each symbol class is a generic symbol. Later symbols in the list often stand for explicit language entities. For example E is reserved for the type constructor of the extensible data type. The concrete symbols are listed in their entirety in Figure 7b.

The target language is the same as GHC Haskell 6.4 with the *glasgow extensions*[4] and *allow undecidable instance* options enabled, modulo the syntactic abbreviations we use. In particular, it has type classes, existential types and allows recursive dictionaries to be created during constraint resolution.

6.2 The rules

The translation is presented in an inductive manner. The "base case" concerns the translation of the *open data* declaration while the inductive step demonstrates the nth extension of the data type and the mth new function on that data type.

We've already introduced the terms *component type* and *functionality class*, but due to their specific meaning they are summarised again.

- *Component type* – A type that forms part of the EDT. There is the *initial component type* which is introduced when translating the *open data* declaration. Then there are the *extension component types* each introduced with the *extend data* declaration.
- *Functionality class* – Classes that provide the functionality for the EDT. There is at most one per module.

There are three indexes, i, j_i and k_i used in the translation.

- The index i ranges over the component types and functionality classes. We have made another presentation simplifying assumption that whenever an extension is made to the open data type that a new function is also declared on the EDT[5].
- Index j_i ranges over the variants (constructors) of the component type and has values $1 \leq j_i \leq n_i$, where n_i is the number of variants for the i_{th} component type
- Index k_i ranges over the functions in a functionality class and has values $1 \leq k_i \leq p_i$, where p_i is the number of functions in the i_{th} functionality class.

$\mathcal{T}^{sort}_{description}$ is the way we denote translation rules. The *sort* is the language entity we are doing the translation on. For instance, $\mathcal{T}^{\sigma}_{method}$ transforms σ-types. Some of the translation rules take arguments e.g.$\mathcal{T}^{e}_{unwrap}$. A translation rule can also be mapped over a sequence; this is denoted $\overline{\mathcal{T}^{sort}_{description}}$.

The translation rules use a form of a pattern matching. Most symbols appearing between the Oxford brackets ($[\![\ldots]\!]$) are generic; they bind to whatever is in their position. However, some symbols are concrete and for a match to occur the symbol in the scrutinee of a translation function must match with the symbol in the pattern. Just like Haskell, a pattern match failure means that a match should be attempted on the next translation rule. A list of the concrete symbols for the source language appears in Figure 7b.

A syntax has been introduced to range over multiple, similar declarations. An expression of the form $\langle\ expression \rangle^m_{j=a}$ means "range over the index j from a to m". There can be nested loops too. An expression of the form $\langle\ expression\ \rangle^{m,n}_{j=a,k=b}$ means that k ranges over b to n for each j. When seen on the left hand side of a translation rule it *matches* on declarations. On the right hand side it *generates* declarations.

Certain information is required by the translation.

- The name of the extensible data type, denoted E in the translation rules.

[4] We do not even require everything that this enables. We only need multi-parameter type classes, scoped type variables, kind annotations and zero constructor data types.

[5] One could always define an identity function or an empty component type if they didn't want one or the other.

E	The extensible data type.
$\mathcal{E}_{i,j_i}$	Constructor of EDT ($0 \leq i \leq m, 1 \leq j_i \leq n_i$).
f_{i,k_i}	Function defined on EDT ($0 \leq i \leq m, 1 \leq k_i \leq p_i$).

Figure 7b. Concrete symbols of the source language

$\mathbf{E}$	Wrapper type for the EDT .
$\mathcal{E}$	Constructor for the wrapper type.
E_i	Component type of EDT.
$\mathcal{E}_{i,j_i}$	Constructor of component type. E_i ($0 \leq i \leq m, 1 \leq j_i \leq n_i$).
S	*Sat* class.
F_i	Functionality class (for functions f_{i,k_i} ($1 \leq k_i \leq p_i$).
P	Proxy type.
d_1	Method of S class. Returns explicit dictionary.
d_i	Selector method for next explicit dictionary in explicit dictionary D_{n-1} ($1 \leq i \leq m$).
D_i	Explicit dictionary for functionality class F_i ($1 \leq i \leq m$).
$\hat{F}_i$	Capping class for functionality class F_i.
$\hat{D}_i$	Capping type for functionality class F_i.
ε_{i,j_i}	Smart constructor for constructor $\mathcal{E}_{i,j_i}$ ($0 \leq i \leq m, 0 \leq j_i \leq n_i$).

Figure 7c. Concrete symbols of the target language

- A collection, $\Gamma(E)$, of all type constructors whose definition directly or indirectly contain occurrences of the type constructor E
- A collection, $\Delta(E)$, of all functions that directly or indirectly contain occurrences of a function, f_i ($i \geq 0$), defined on the EDT, $E\ \overline{\alpha}$.

For example, an analysis on the following module would yield $\Gamma(E) = \{T, T'\}, \Delta(E) = \{g, h\}$.

$$\begin{array}{l}
\textbf{open data}\ E\ a\ =\ \ldots \\
\\
data\ T\ b\ c\ =\ T_1\ b\ (T'\ c) \\
data\ T'\ a\ =\ T'_1\ (E\ a) \\
\\
f\ ::\ E\ a\ \rightarrow\ a \\
f\ =\ \ldots \\
\\
g\ =\ \ldots\ f\ \ldots \\
h\ =\ \ldots\ g\ \ldots
\end{array}$$

The translation of a module containing an *extend data* requires additional information but we defer discussion of this until Section 6.4.

6.3 Base case: Translating *open data*

A portion of the rule used to translate *open data* declarations appears in Figure 8a. The complete rules appear in Figures 10a and 10b. The portion provided introduces the initial functionality class F_0, and an instance for the first functions on the EDT, $f_{0,k}$ (where $1 \leq k \leq p_0$). A capping class, $\hat{F}_0$, and capping type, $\hat{D}_0$ are also introduced. (There is no explicit dictionary for the base functionality class.) The complete rule also introduces the initial component type, E_0, and corresponding smart constructors $\varepsilon_{0,j}$ (for $1 \leq j \leq n_0$), the proxy type P, the wrapper type $\mathbf{E}$ and a corresponding unwrapping instance. The *Sat* class, S is also introduced, once and for all.

Smart constructors are introduced so that the translation of regular data constructors in the source language is simplified; an

$$\begin{array}{l}
\mathcal{T}^{data}\llbracket\, \textbf{open data}\ E\ \overline{\alpha}\ =\ \langle\ \mathcal{E}_{0,j_0}\ \overline{\tau_j}\ \rangle_{j=1}^{n_0}; \\
\qquad \langle\, f_{0,k_0}\ :\ \sigma_{0,k_0}\ \rangle_{k_0=1}^{p_0}; \\
\langle\, f_{0,k_0}\ (\ \mathcal{E}_{0,j_0}\ \overline{\nu_{j_0}}\ :\ E\ \overline{\xi_{j_0}}\ )\ =\ b_{0,j_0,k_0}\ \rangle_{j=1,k_0=1}^{n_0,p_0}\ \rrbracket\ = \\
\\
\quad \ldots \\
\textsf{Initial functionality class} \\
\quad \textbf{class}\ S\ (\delta\ \beta)\ \Rightarrow\ F_0\ \delta\ \beta\ \textbf{where} \\
\qquad \langle\, f_{0,k_0}\ :\ \mathcal{T}^{\sigma}_{method}(0,k_0)\llbracket\sigma_{0,k_0}\rrbracket\ \rangle_{k=1}^{p_0}\ ; \\
\textsf{Initial functionality instance} \\
\quad \textbf{instance}\ (S\ (\delta\ (\mathbf{E}\ \delta)),\ S\ (\delta\ (E_0\ \delta))) \\
\qquad\qquad\qquad \Rightarrow\ F_0\ \delta\ (E_0\ \delta)\ \textbf{where} \\
\qquad \langle\, f_{0,k_0}\ (_ : P\ \delta)\ (E_0\ \overline{\nu_{j_0}}\ :\ E_{0,j_0}\ \delta\ \overline{\xi_{j_0}})\ = \\
\qquad\quad \mathcal{T}^{e}_{method}\llbracket b_{0,j_0,k_0}\rrbracket\ \rangle_{j_0=1,k_0=1}^{n_0,p_0} \\
\textsf{Capping class, type and instances} \\
\quad \textbf{data}\ \hat{D}_0\ \beta\ ; \\
\quad \textbf{class}\ F_0\ \hat{D}_0\ \beta\ \Rightarrow\ \hat{F}_0\ \beta\ ; \\
\quad \textbf{instance}\ \hat{F}_0\ (\mathbf{E}\ (\hat{D}_0))\ ; \\
\quad \textbf{instance}\ \hat{F}_0\ (E_0\ (\hat{D}_0))\ ; \\
\quad \textbf{instance}\ \hat{F}_0\ \beta\ \Rightarrow\ S\ (\hat{D}_0\ \beta)\ \textbf{where} \\
\qquad d_1\ =\ \bot \\
\quad \ldots
\end{array}$$

Figure 8a. A portion of the translation for *open data* declaration in the initial module ($m = 0$).

$$\begin{array}{l}
\mathcal{T}^{data}\llbracket\, \langle\, f_{0,k_0}\ (\ \mathcal{E}_{m,j_m}\ \overline{\nu_{j_m}}\ :\ E\ \overline{\xi_{j_m}})\ =\ b_{0,j_m,k_0}\ \rangle_{j_m=1,k_0=1}^{n_m,p_0}; \\
\qquad \ldots\ ; \\
\qquad \langle\, f_{m-1,k_{m-1}}\ (\ \mathcal{E}_{m,j_m}\ \overline{\nu_{j_m}}\ :\ E\ \overline{\xi_{j_m}})\ = \\
b_{m-1,j_m,k_{m-1}}\ \rangle_{j_m=1,k_{m-1}=1}^{n_m,p_{m-1}}\ \rrbracket\ = \\
\quad \langle\ \textbf{instance}\ (\ S\ (\overline{D}^m \delta_m\ (\mathbf{E}\ (\overline{D}^m \delta_m))) \\
\qquad\qquad\qquad ,\ S\ (\overline{D}^m \delta_m\ (E_0\ (\overline{D}^m \delta_m))) \\
\qquad\qquad\qquad ,\ \ldots \\
\qquad\qquad\qquad ,\ S\ (\overline{D}^m \delta_m\ (E_m\ (\overline{D}^m \delta_m)\ )) \\
\qquad\qquad\qquad)\ \Rightarrow\ F_i\ (\overline{D}_i^m \delta_m)\ (E_m\ (\overline{D}^m \delta_m))\ \textbf{where} \\
\qquad \langle\, f_{i,k_i}\ (_ : P\ (\overline{D}^m \delta_m)) \\
\qquad\qquad (\ \mathcal{E}_{m,j_m}\ \overline{\nu_{j_m}}\ :\ E_m\ (\overline{D}^m \delta_m)\ \overline{\xi_{j_m}})\ = \\
\qquad\qquad\quad \mathcal{T}^{e}_{method}\llbracket b_{i,j_m,k_i}\rrbracket\ \rangle_{j_m=1,k_i=1}^{n_m,p_i} \\
\rangle_{i=0}^{m-1}
\end{array}$$

Figure 8b. Translation for new equations on existing functions in the m_{th} extension module.

occurrence of a constructor becomes a smart constructor instead. An extra argument of the proxy type is added for all functions, f_{i,k_i}, defined on the EDT and to the smart constructors.

6.4 Inductive step: Translating *extend data*

The portion of the rules for translating a module containing an *extend data* declaration appears in Figures 8b and 8c. The complete rules appear in 10b, 10c, and 10d.

These rules introduce the mth new variant on the EDT and the mth function. It is assumed that the following information is available.

- A list of m existing functionality classes $[F_0, ..., F_{m-1}]$, functions $[f_{0,k_0}, \ldots, f_{m-1,k_{m-1}}]$ (where $1 \leq k_i \leq p_i$) and explicit dictionaries $[D_0, \ldots, D_{m-1}]$.

$$
\begin{array}{l}
\mathcal{T}^{data}\llbracket\, \mathbf{extend\ data}\ E\ \overline{\alpha}\ =\ \langle\ \mathcal{E}_{m,j}\ \overline{\tau_j}\ \rangle_{j_m=1}^{n_m}\ ; \\
\quad \langle\, f_{m,k_m}\ :\ \sigma_{m,k_m}\ \rangle_{k_m=1}^{p_m}; \\
\quad \langle\, f_{m,k_m}\ (\ \mathcal{E}_{m,j_m}\ \overline{\nu_{m,j_m}}\ :\ E\ \overline{\xi_{m,j_m}})\ = \\
\qquad b_{m,j_m,k_m}\ \rangle_{j_m=1,k_m=1}^{n_m,p_m}\ \rrbracket\ =
\end{array}
$$

$\ldots$

The m_{th} functionality class

$$
\begin{array}{l}
\mathbf{class}\ (S\ (\overline{D}^m \delta_m\ \beta),\ F_{m-1}\ (D_m\ \delta_m)\ \beta) \\
\qquad \Rightarrow\ F_m\ \delta_m\ \beta\ \mathbf{where} \\
\quad \langle\, f_{m,k_m}\ :\ \mathcal{T}^{\sigma}_{method}\llbracket \sigma_{m,k_m} \rrbracket\ \rangle_{k_m=1}^{p_m};
\end{array}
$$

The m_{th} explicit dictionary

$$
\begin{array}{l}
\mathbf{data}\ D_m\ \delta_m\ \beta\ = \\
\quad \mathcal{D}_m\ \{\ \langle\, f'_{m,k_m}\ :\ \mathcal{T}^{\sigma}_{method}\llbracket \sigma_{m,k_m} \rrbracket;\ \rangle_{k_m=1}^{p_m}\ ; \\
\qquad ,\ d_{m+1}\ :\ (\delta_m\ \beta)\ \}
\end{array}
$$

Functionality instances (for component types $0 \leq i \leq m$)

$$
\begin{array}{l}
\langle\ \mathbf{instance}\ (\ S\ (\overline{D}^m \delta_m\ (\mathbf{E}\ (\overline{D}^m \delta_m)\ )) \\
\qquad\qquad ,\ S\ (\overline{D}^m \delta_m\ (E_0\ (\overline{D}^m \delta_m)\ )) \\
\qquad\qquad ,\ \ldots \\
\qquad\qquad ,\ S\ (\overline{D}^m \delta_m\ (E_m\ (\overline{D}^m \delta_m)\ )) \\
\qquad\qquad)\ \Rightarrow\ F_m\ \delta_m\ (E_i\ (\overline{D}^m \delta_m)\ )\ \mathbf{where} \\
\quad \langle\, f_{m,k_m}\ (\_ : P\ (\overline{D}^m \delta_m))\ (\ \mathcal{E}_{i,j_i}\ \overline{\nu_{j_i}}\ :\ E_i\ (\overline{D}^m \delta_m)\ \overline{\xi_{j_i}})\ = \\
\quad \mathcal{T}^{e}_{method}\llbracket b_{i,j_i,k_m} \rrbracket\ \rangle_{j_i=1,k_m=1}^{n_i,p_m} \\
\rangle_{i=0}^{m}
\end{array}
$$

Capping class, type and instances

$$
\begin{array}{l}
\mathbf{data}\ \hat{D}_m\ \beta\ ; \\
\mathbf{class}\ F_m\ \hat{D}_m\ \beta\ \Rightarrow\ \hat{F}_m\ \beta\ ; \\
\mathbf{instance}\ \hat{F}_m\ (\mathbf{E}\ (\overline{D}^m\ \hat{D}_m))\ ; \\
\langle\ \mathbf{instance}\ \hat{F}_m\ (E_i\ (\overline{D}^m\ \hat{D}_m))\ ;\ \rangle_{i=0}^{m}
\end{array}
$$

"Knot tying" instance

$$
\begin{array}{l}
\mathbf{instance}\ \hat{F}_m\ \beta\ \Rightarrow\ S\ (\overline{D}^m\ \hat{D}_m\ \beta)\ \mathbf{where} \\
d_1\ =\ D_1\ \{\ \langle\, f'_{1,k}\ =\ f_{1,k}\ \rangle_{k=1}^{p_1} \\
\qquad\qquad d_2\ =\ D_2\ \{\ \langle\, f'_{2,k}\ =\ f_{2,k},\ \rangle_{k=1}^{p_2} \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad d_m\ =\ D_m\ \{\ \langle\, f'_{m,k}\ =\ f_{m,k}\ \rangle_{k=1}^{p_m}, \\
\qquad\qquad\qquad\qquad\qquad\quad d_{m+1}\ =\ \perp\}\ \ldots\ \} \\
\ldots
\end{array}
$$

Figure 8c. A portion of the translation for *extend data* declaration and new function for the m_{th} extension module.

- A list of m existing component types $[E_0, ..., E_{m-1}]$ and the variant constructors $[\mathcal{E}_{0,j_0}, \ldots, \mathcal{E}_{m-1,j_{m-1}}]$ (where $1 \leq j_i \leq n_i$) .
- A list of capping classes, $[\hat{F}_1, \ldots, \hat{F}_{m-1}]$ and capping types, $[\hat{D}_1, \ldots, \hat{D}_{m-1}]$. A capping type is just a zero constructor dummy type.

Similar to the base case, the rule in Figure 8c introduces a new component type and smart constructor, a new functionality class, function, and capping class. An instance is introduced for each existing component type and the newly introduced one.

Also, the rule presented in Figure 8b introduces instances to handle new equations on old functions (i.e. f_{i,k_i} $(i\ <\ m,\ 1\ \leq\ k_i\ \leq\ p_i)$). (Remember, there is a syntactic restriction on the source language specifying that these must have been declared.) This rule also brings into being an explicit dictionary and associated capping type.

In many ways the inductive step of the translation is more interesting. Consequently we spend some time explaining the subtleties of the rules.

$$
\begin{array}{l}
\mathcal{T}^{\overline{\alpha}}_{kind}\llbracket \alpha_1,\ \ldots,\ \alpha_k \rrbracket\ =\ \overbrace{(\star\ \rightarrow\ \ldots\ \rightarrow\ \star)}^{k+1}\ \rightarrow\ \star \\
\mathcal{T}^{\sigma}_{method}\llbracket\ \forall\overline{\alpha}.E\ \overline{\tau} \\
\qquad \rightarrow\ \xi\ \rrbracket\ =\ \forall\overline{\alpha}.\ P(\overline{D}^m \delta_m)\ \rightarrow\ \beta\ \overline{\tau}\ \rightarrow\ \mathcal{T}^{\tau}_{method}\llbracket \xi \rrbracket \\
\mathcal{T}^{\tau}_{method}\llbracket \alpha \rrbracket\ =\ \alpha \\
\mathcal{T}^{\tau}_{method}\llbracket T \rrbracket\ =\ \begin{cases} T\ (\overline{D}^m \delta_m) & \text{, if } T \in \Gamma(E) \\ T & \text{, otherwise} \end{cases} \\
\mathcal{T}^{\tau}_{method}\llbracket E \rrbracket\ =\ E\ (\overline{D}^m \delta_m) \\
\mathcal{T}^{\tau}_{method}\llbracket \tau_1\ \tau_2 \rrbracket\ =\ \mathcal{T}^{\tau}_{method}\llbracket \tau_1 \rrbracket\ \mathcal{T}^{\tau}_{method}\llbracket \tau_2 \rrbracket \\
\\
\mathcal{T}^{e}_{method}\llbracket f_{i,k_i} \rrbracket\ =\ \begin{cases} f'_{i,k_i}\ \mathcal{T}^{dict}(i)\ (\perp : P\ (\overline{D}^m \delta_m)) \\ \text{, if } i > 0 \\ f_{i,k_i}\ (\perp : P\ (\overline{D}^m \delta_m)) \\ \text{, otherwise} \end{cases} \\
\mathcal{T}^{e}_{method}\llbracket x \rrbracket\ =\ \begin{cases} x\ (\perp : P\ (\overline{D}^m \delta_m)) & \text{, if } x \in \Delta(E) \\ x & \text{, otherwise} \end{cases} \\
\mathcal{T}^{e}_{method}\llbracket \lambda x\ :\ \tau.e \rrbracket\ =\ \lambda x : \mathcal{T}^{\tau}_{method}\llbracket \tau \rrbracket.\ \mathcal{T}^{e}_{method}\llbracket e \rrbracket \\
\mathcal{T}^{e}_{method}\llbracket\ \mathcal{E}_{i,j_i} \rrbracket\ =\ \varepsilon_{i,j_i}\ (\perp :\ P\ (\overline{D}^m \delta_m)) \\
\mathcal{T}^{e}_{method}\llbracket\ \mathcal{C} \rrbracket\ =\ \mathcal{C} \\
\mathcal{T}^{e}_{method}\llbracket e_1\ e_2 \rrbracket\ =\ \mathcal{T}^{e}_{method}\llbracket e_1 \rrbracket \mathcal{T}^{e}_{method}\llbracket e_2 \rrbracket \\
\\
\mathcal{T}^{dict}(i)\ =\ (d_i\ (\ldots\ (d_2\ d_1)\ \ldots))
\end{array}
$$

Figure 8d. A portion of the translation rules

S	Sat	F_1	$Eval$
δ_i	cxt	$f_{1,1}$	$eval$
β	b	$f_{1,2}$	$apply$
$\mathbf{E}$	Exp	$\hat{F}_0$	$AlphaCap$
E_0	Exp_0	$\hat{D}_0$	$AlphaEnd$
F_0	$Alpha$	d_1	$dict$
$\mathcal{E}$	$MkExp$	d_2	$expExt$
$\mathcal{E}_{0,1}$	Var	D_1	$Eval$
$\mathcal{E}_{0,2}$	Lam	$\hat{F}_1$	$EvalCap$
$\mathcal{E}_{0,3}$	App	$\hat{D}_1$	$EvalEnd$
$f_{0,1}$	$alpha$	$(n_0\ =\ 3,\ p_0\ =\ 1,$	$n_1\ =\ 1,\ p_1\ =\ 2)$
$\mathcal{E}_{1,1}$	$LetE$	$(m = 2)$	
E_1	Exp_1		

Figure 9. A mapping from symbols in the formal translation to identifiers in the running example.

6.4.1 The need for proxy arguments

Proxy arguments guide the type checker for the target language. Consider the following function in the source language:

$$
\begin{array}{l}
\mathbf{data}\ E\ =\ \mathcal{E}_0\ String\ (E\ String) \\
f_{0,1}\ ::\ E\ \rightarrow\ String \\
f_{0,1}\ (\ \mathcal{E}_0\ s\ e)\ =\ s\ +\!\!+\ f_{0,1}\ e
\end{array}
$$

Now consider what we would get if the translation omitted to add proxy arguments.

$$
\begin{array}{l}
\mathbf{class}\ S\ (\ \delta\ \beta)\ \Rightarrow\ F_0\ \delta\ \beta\ \mathbf{where} \\
\quad f_0\ ::\ \beta\ \rightarrow\ String \\
\\
\mathbf{instance}\ S\ (\delta\ (\ \mathbf{E}\ \delta))\ \Rightarrow\ F_0\ \delta(\ \mathbf{E}\ \delta)\ \mathbf{where} \\
\quad f_{0,1}\ (\ \mathcal{E}\ x)\ =\ f_{0,1}\ x \\
\mathbf{instance}\ (S\ (\delta\ (\ \mathbf{E}\ \delta)),\ S\ (\delta\ (E_0\ \delta)))\ \Rightarrow\ F_0\ \delta\ (E_0\ \delta)\ \mathbf{where} \\
\quad f_{0,1}\ (\ \mathcal{E}_0\ s\ e)\ =\ s\ +\!\!+\ f_{0,1}\ e
\end{array}
$$

Among the constraints raised by the use of $f_{0,1}$ on the right hand side of the instance method equation is $F_0\ \delta'\ (\ \mathbf{E}\ \delta)$. The problem is that the δ' and δ aren't equal. The proxy ensures that they are equated. To see this consider the translation with proxy arguments attached.

$$\begin{array}{l}\mathbf{class}\ S\ (\ \delta\ \beta)\ \Rightarrow\ F_0\ \delta\ \beta\ \mathbf{where}\\ \quad f_{0,1}\ ::\ P\ \delta\ \rightarrow\ \beta\ \rightarrow\ \mathit{String}\end{array}$$

$$\begin{array}{l}\mathbf{instance}\ (S\ (\delta\ (\ \mathbf{E}\ \delta)),\ S\ (\delta\ (E_0\ \delta)))\ \Rightarrow\ F_0\ \delta\ (E_0\ \delta)\ \mathbf{where}\\ \quad f_{0,1}\ (\_\ :\ P\ \delta)\ (\ \mathcal{E}_0\ s\ e)\ =\ s\ +\!\!+\ f_{0,1}\ (\bot\ :\ P\ \delta)\ e\end{array}$$

The constraint raised by the expression $f_{0,1}\ (\bot\ :\ P\ \delta)\ e$ is now $F_0\ \delta\ (\ \mathbf{E}\ \delta)$.

6.4.2 S constraints in instance heads

The instance heads for new equations on existing component types and the instance heads for new functions both contain many occurrences of S constraints. This may seems strange considering that each functionality class has S as a superclass. The reason is that the S instance that "ties the knot" will be declared at some point in the future (possibly in another module). The S constraints in the instance head "promise" that this will happen.

These constraints mention the latest explicit dictionary (i.e. $\overline{D}^m$). The purpose of this is to allow the body of the instance method to contain occurrences of any of the functions so far $(f_{1,k_1}, \ldots, f_{m-1,k_{m-1}})$ *and* the latest ones – f_{m,k_m}). This is possible even inside new equations on existing functions, which may seem counter-intuitive at first. To see why consider the translation of the following new equation where $a < m$, $b <= m$, and $a < b$.

$$\begin{array}{l}\mathcal{T}^{data}[\![f_{a,1}\ (\ \mathcal{E}_{m,2}\ x)\ =\ \ldots\ f_{b,1}\ \ldots]\!]\ =\\ \ldots\\ \mathbf{instance}\ (\ S\ (\overline{D}^m\ \delta_m\ (\ \mathbf{E}\ (\overline{D}^m\ \delta_m)))\\ \qquad\qquad ,\ S\ (\overline{D}^m\ \delta_m\ (E_0\ (\overline{D}^m\ \delta_m)))\\ \qquad\qquad ,\ \ldots\\ \qquad\qquad ,\ S\ (\overline{D}^m\ \delta_m\ (E_m\ (\overline{D}^m\ \delta_m)))\\ \qquad\qquad)\ \Rightarrow\ F_a\ (\overline{D}^m_{a+1}\ \delta_m)\ E_m\ (\overline{D}^m\ \delta_m)\ \mathbf{where}\\ f_{a,1}\ (\ \mathcal{E}_{m,2}\ x)\ =\ \ldots\ f'_{b,1}\ (d_m\ \ldots\ (d_2\ d_1)\ldots)\ \ldots\\ \ldots\end{array}$$

The expression $f'_{b,1}\ (d_b\ \ldots\ (d_2\ d_1)\ldots)$ raises the following constraints.

$$\begin{array}{l}\quad S\ (\overline{D}^m\ \hat{D}_m\ (E_m\ (\overline{D}^m\ \hat{D}_m)))\\ \rightsquigarrow\ \hat{F}_m\ (E_m\ (\overline{D}^m\ \hat{D}_m))\end{array}$$

This instance for the capping class has been declared. The considerably involved way in which this is type checked is covered in the next section.

6.4.3 Capping classes

Instances of the capping class, and the associated S instance, are used to "tie the knot" during constraint resolution. They do this, not just for the m_{th} functionality class, but for all the others.

For each of the capping class instances we need to check for the existence of an instance of its super class, the m_{th} functionality class. Because constraint resolution is cycle aware we first add the constraints $\hat{F}_m\ (E_i\ (\overline{D}^m \hat{D}_m))$ (for $0\ \leq\ i\ \leq\ m$) to the current collection of assumptions. (Each of these constraints will only be resolved if a chain of resolutions reaches it again.) Now let's consider a particular superclass constraint for component type E_b (for some $0 \leq b \leq m$). It produces $m+1$ S constraints.

$$\begin{array}{l}\quad F_m\ \hat{D}_m\ (E_b\ (\overline{D}^m\ \hat{D}_m))\\ \rightsquigarrow\ S\ (\overline{D}^m\ \hat{D}_m\ (E_j\ (\overline{D}^m\ \hat{D}_m)))\ \text{for each}\ (0 \leq j \leq m)\end{array}$$

Each one of these S constraint is resolved by

$$\begin{array}{l}\quad S\ (\overline{D}^m\ \hat{D}_m\ (E_j\ (\overline{D}^m\ \hat{D}_m)))\\ \rightsquigarrow\ \hat{F}_m\ (E_j\ (\overline{D}^m\ \hat{D}_m))\end{array}$$

But these are in the collection of assumptions, so they get resolved. Thus, a recursive dictionary is created for each component type and the wrapper type. This "ties the knot" for *all* of the functionality classes, not just the m_{th} one. To see why, consider how we type check

$f'_{a,c}\ (d_i\ \ldots\ (d_2\ d_1)\ldots)\ ::\ E_b\ (\overline{D}^m\ \hat{D}^m)\ \rightarrow\ \ldots$
for some $0 \leq a \leq m$, $0 \leq b \leq m$, and $0 \leq c \leq p_i$. This leads to the following constraint resolution. (The initial constraint comes from substituting $\delta\ =\ \overline{D}^m_{i+1}\ \hat{D}_m$ into $S\ (\overline{D}^i\ \delta\ (E_b\ (\overline{D}^i\ \delta)))$.

$$\begin{array}{l}\quad S\ (\overline{D}^m\ \hat{D}_m\ (E_b\ (\overline{D}^m\ \hat{D}_m)))\\ \rightsquigarrow\ \hat{F}_m\ (E_b\ (\overline{D}^m\ \hat{D}_m))\end{array}$$

But this constraint has been provided by the capping class instance which type checks for the reasons stated earlier.

6.4.4 Translating regular declarations

Any type declaration in the source language that directly or indirectly contains a reference to E must be translated to contain an occurrence of the wrapper data type $\mathbf{E}$ in the target language. Any function which directly or indirectly contains an occurrence of a function $f_{i,k}$ (for a specific $0\ \leq\ i\ \leq\ m, 0\ \leq\ k\ \leq\ p_i$) must also have its body transformed to contain an occurrence of $f'_{i,k}\ d_i\ (\ldots\ (d_2\ d_1)\ldots)$. More importantly, an S constraint must be added to the type. However, this only needs to be done for the wrapper data type, $\mathbf{E}$, as these are the only values that will be passed to such functions. S constraints on component types are only ever seen in the class and instance heads of functionality classes.

6.5 The link between the formalisation and the running example

To further the reader's understanding they are encouraged to apply the rules from Figures 10a and 10b to the initial module of the running example (Figure 1) to yield the result in Figures 4a–4g. By applying the rules in Figures 10c, 10d and 10b to to Figure 2 they will get the result in Figures 5a–5h.

However, the translation rules use an abbreviated syntax. In order to aid the reader Figure 9 shows the correspondence between the abbreviated syntax and the syntax used in the translation of the running example.

6.6 Creating values of the EDT

For an extensible data type to be created and used, it is necessary for the translator to insert the latest capping type in place of the δ_m in the type $\mathbf{E}\ (\overline{D}^m\ \hat{D}_m)\ \overline{\tau})$. However, the only functions that can be called on a value of this type are those that have the constraint $S\ (\overline{D}^a\ \delta_a\ (\ \mathbf{E}\ (\overline{D}^a\ \delta_a)))$ (where $a\ \leq\ m$) in their types. The translation ensures that the functions f_{i,k_i} (where $0\ \leq\ i\ \leq\ m, 1\ \leq\ k_i\ \leq\ p_i$) (and any function which directly or indirectly calls them) satisfy this condition.

7. Related work

To date, the only published (Haskell) solution to the expression problem is Löh and Hinze [9]. They describe a method whereby the amount of recompilation can be kept to a minimum. This is a two tiered solution. First, the open declarations and closed declarations are separated out into the *Main* module. Unfortunately, this often results in a mutually dependency with each module that contained open declarations. Although the modules can be compiled separately they cannot be re-compiled independently; a change to an open entity necessitates recompilation of all modules depending on *Main*. Next, the left and right hand sides of the open equations are separated into two new equations. The first does the pattern matching and dispatches to the second which is moved back to the module it was originally declared in. As long as the interface between this module and the *Main* module remains stable a change to open entities only results in a re-compilation of the *Main* module. This fact disqualifies the solution from achieving true separate compilation. We believe that open data types are eminently useful in plug-in enabled applications. It is unclear how well Löh and Hinze's solution works in a plug-in environment. It may be possible to use Stewart and Chakravarty's [12] method to re-load the entire application but this seems much more complicated that our solution and would require loading the entire program not just the plug-in module.

A number of informal type-classed based (e.g. [6]) have been proposed. However, there is a crucial difference with our solution. Where these solutions lift constructor values to the type level, ours does not. This means that functions can still be written in a natural way using the full power of Haskell's pattern matching. Also, there is still a clear relation between a constructor and the data type it creates; a constructor creates values of its component type.

Another notable solution to the expression problem is provided by Kiselyov and Lämmel [4]. This requires that programs be written in an object oriented style. In our solution, functions on open data types are merely overloaded functions and the construction of values by smart constructors is almost as natural as with regular constructors.

Several papers ([15], [5], [10], [1]) have focused on extending object-oriented languages in order to make the addition of extra functionality easier. (Of these, only Zenger and Odersky's and Bruce's solutions can be statically type checked.) However, we wish to do the converse by making the addition of variants easier in a functional language. Solutions in functional languages have also been studied. Solutions have been proposed in OCaml [2] and the hybrid object-oriented/functional language, Scala [16]. The solutions in OCaml and Scala both use a notion of sub-typing. OCaml provides this through *polymorphic variants*—constructors that can belong to more than one data type. Mixins are used in Scala.

8. Conclusion

We have presented a solution to the expression problem which provides true separate compilation and works in current implementations of Haskell. The main ingredients of the solution are multi-parameter type classes, existential types and recursive dictionaries. A formal translation has been provided that can be used as the basis of an pre-processor implementation. However, the technique is readily usable as a programming idiom.

The source code of the examples in this paper can be found as a *darcs* repository at:

`http://www.cse.unsw.edu.au/~sseefried`
`/code/exp_prob`

9. Acknowledgements

The authors would like to thank Roman Leshchinskiy for invaluable help in improving the presentation of the translation.

References

[1] Kim B. Bruce. Some Challenging Typing Issues in Object-Oriented Languages: Extended Abstract. In , volume 82.8 of *Electronic Notes in Theoretical Computer Science*, pages 1–29, 2003.

[2] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, November 2000.

[3] R.J.M Hughes. Restricted data types in Haskell. In *Proceedings of the 1999 Haskell Workshop*, 1999.

[4] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. 2005.

[5] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. *Lecture Notes in Computer Science*, 1445:91–??, 1998.

[6] Ralf Lämmel. *Extensible grammars*, on the `comp.compilers` newsgroup, `http://compilers.iecc.com/comparch/article/04-12-111`, 2004.

[7] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*. ACM Press, September 2005.

[8] Konstantin Läufer. Type Classes with Existential Types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[9] Andres Löh and Ralf Hinze. Open data types and open functions. In *PPDP'06: Eighth ACM-SIGPLAN International Symposium on Principles and Practice of Declaritive Programming*, Venice, Italy, July 2006.

[10] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proc. 12th International Conference on Compiler Construction*, number 2622 in Lecture Notes in Computer Science, pages 138–152. Spring-Verlag, April 2003.

[11] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In S. A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168, 1975.

[12] Don Stewart and Manuel M. T. Chakravarty. Dynamic Applications From the Ground Up. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM Press, September 2005.

[13] Philip Wadler. *The expression problem*, Discussion on the Java Genericity mailing list, 1998.

[14] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Conference Record of the Sixteenth Annual Symposium on Principles of Programming Languages (POPL'89)*, pages 60–76, Austin, Texas, January 1989. ACM Press.

[15] Matthias Zenger and Martin Odersky. Extensible Data Types with Defaults. In *International Conference on Functional Programming (IFCP'01)*, pages 241–252, Firenze, Italy, September 2001.

[16] Matthias Zenger and Marting Odersky. Independently Extensible Solutions to the Expression Problem. Technical Report IC/2004/33, École Polytechnique Fédérale de Lausanne, Lausanna, Switzerland, 2004.

A. Translation rules

$$\mathcal{T}^{data}[\![\, \textbf{open data}\ E\ \overline{\alpha}\ =\ \langle\ \mathcal{E}_{0,j_0}\ \overline{\tau_j}\ \rangle_{j=1}^{n_0};$$
$$\langle\ f_{0,k_0}\ :\ \sigma_{0,k_0}\ \rangle_{k_0=1}^{p_0};\ \langle\ f_{0,k_0}\ (\ \mathcal{E}_{0,j_0}\ \overline{\nu_{j_0}}\ :\ E\ \overline{\xi_{j_0}}\ )\ =\ b_{0,j_0,k_0}\ \rangle_{j=1,k_0=1}^{n_0,p_0}\ ]\!]\ =$$

Proxy type

$\textbf{data}\ P\ \delta\ ;$

Sat class

$\textbf{class}\ S\ \alpha\ \textbf{where}$
$\quad d_1\ :\ \alpha$

Wrapper type

$\textbf{data}\ \mathbf{E}\ (\delta\ :\ \mathcal{T}_{kind}^{\overline{\alpha}}[\![\overline{\alpha}]\!])\ \overline{\alpha}\ =\ \exists\beta.F_0\ \delta\ \beta\ \Rightarrow\ \mathcal{E}\ (\beta\ \overline{\alpha});$

Initial component type

$\textbf{data}\ E_0\ \delta\ \overline{\alpha}\ =\ \langle\ \mathcal{E}_{0,j_0}\ \overline{\mathcal{T}_E^{\tau}}[\![\overline{\tau_{j_0}}]\!]\ \rangle_{j_0=1}^{n_0}\ ;$

Initial functionality class

$\textbf{class}\ S\ (\delta\ \beta)\ \Rightarrow\ F_0\ \delta\ \beta\ \textbf{where}$
$\quad \langle\ f_{0,k_0}\ :\ \mathcal{T}_{method}^{\sigma}(0,k_0)[\![\sigma_{0,k_0}]\!]\ \rangle_{k=1}^{p_0}\ ;$

Initial functionality instance

$\textbf{instance}\ (S\ (\delta\ (\mathbf{E}\ \delta)),\ S\ (\delta\ (E_0\ \delta)))\ \Rightarrow\ F_0\ \delta\ (E_0\ \delta)\ \textbf{where}$
$\quad \langle\ f_{0,k_0}\ (\_ : P\ \delta)\ (E_0\ \overline{\nu_{j_0}}\ :\ E_{0,j_0}\ \delta\ \overline{\xi_{j_0}})\ =\ \mathcal{T}_{method}^{e}[\![b_{0,j_0,k_0}]\!]\ \rangle_{j_0=1,k_0=1}^{n_0,p_0}\ ;$

Unwrapping instance

$\textbf{instance}\ S\ (\delta\ (\mathbf{E}\ \delta))\ \Rightarrow\ F_0\ \delta\ (\mathbf{E}\ \delta)\ \textbf{where}$
$\quad \langle\ f_{0,k_0}\ (\_ : P\ \delta)\ (\ \mathcal{E}\ x\ :\ \mathbf{E}\ \delta\ \overline{\xi})\ =\ \mathcal{T}_{unwrap}^{\sigma}(f_{0,k_0},\ x, P\ \delta)[\![\sigma_{0,k_0}]\!]\ \rangle_{k=1}^{p_i}$

Capping class, type and instances

$\textbf{data}\ \hat{D}_0\ \beta\ ;$
$\textbf{class}\ F_0\ \hat{D}_0\ \beta\ \Rightarrow\ \hat{F}_0\ \beta\ ;$
$\textbf{instance}\ \hat{F}_0\ (\mathbf{E}\ (\hat{D}_0))\ ;$
$\textbf{instance}\ \hat{F}_0\ (E_0\ (\hat{D}_0))\ ;$
$\textbf{instance}\ \hat{F}_0\ \beta\ \Rightarrow\ S\ (\hat{D}_0\ \beta)\ \textbf{where}$
$\quad d_1\ =\ \bot$

Smart constructors

$\langle\ \varepsilon_{0,j_0} : \mathcal{T}_{smart}^{\overline{\tau}}[\![\overline{\tau_{j_0}}]\!];\ \varepsilon_{0,j_0}\ (\_ : P\ \delta)\ =\ \mathcal{T}_{smart}^{e}(0,j_0)[\![\overline{\tau_{j_0}}]\!]\ \rangle_{j_0=1}^{n_0}$

Figure 10a. Translation for *open data* declaration in the initial module ($m = 0$).

$$
\begin{aligned}
&\mathcal{T}^{data}[\![\mathbf{data}\; T\,\overline{\alpha} \;=\; \mathcal{C}\,\overline{\tau}]\!] = \begin{cases} \mathbf{data}\, T\,\delta\,\overline{\alpha} \;=\; \mathcal{C}\,\overline{\mathcal{T}^{\tau}_{E}[\![\overline{\tau}]\!]} & \text{, if } T \;\in\; \Gamma(E) \\ \mathbf{data}\, T\,\overline{\alpha} \;=\; \mathcal{C}\,\overline{\tau} & \text{, otherwise} \end{cases} \\
&\mathcal{T}^{data}[\![\mathbf{type}\; T\,\overline{\alpha} \;=\; \overline{\tau}]\!] \quad = \overline{\mathcal{T}^{\tau}_{E}[\![\overline{\tau}]\!]} \\
&\mathcal{T}^{tval}[\![x \;:\; \sigma;\; \overline{val}]\!] \quad = x \;:\; \mathcal{T}^{\sigma}_{E}[\![\sigma]\!]\;;\; \overline{\mathcal{T}^{val}[\![\overline{val}]\!]} \\
&\mathcal{T}^{val}[\![x\;p \;=\; e]\!] \quad = \begin{cases} x\;(\_ : P\;(\overline{D}^{m}\delta_m)\;p \;=\; \mathcal{T}^{e}_{method}[\![e]\!] & \text{, if } x \in \Delta(E) \\ x\,p \;=\; e & \text{, otherwise} \end{cases} \\
&\mathcal{T}^{val}[\![x \;=\; e]\!] \quad = \begin{cases} x\;(\_ : P\;(\overline{D}^{m}\delta_m) \;=\; \mathcal{T}^{e}_{method}[\![e]\!] & \text{, if } x \in \Delta(E) \\ x \;=\; e & \text{, otherwise} \end{cases} \\
&\mathcal{T}^{\sigma}_{E}[\![\forall\,\overline{\alpha}.\tau]\!] \quad = \begin{cases} \forall\,\overline{\alpha}.\; S\;(\overline{D}^{m}\delta_m\;(\;\mathbf{E}\;(\overline{D}^{m}\delta_m)))\;\Rightarrow & \\ \quad \mathcal{T}^{\tau}_{E}[\![\tau]\!] & \text{, if } x \in \Delta(E) \\ \mathcal{T}^{\tau}_{E}[\![\tau]\!] & \text{, otherwise} \end{cases}
\end{aligned}
$$

Figure 10b. Translation for regular declarations in the m_{th} module.

$$
\begin{aligned}
&\mathcal{T}^{data}[\![\;\langle\, f_{0,k_0}\;(\;\mathcal{E}_{m,j_m}\;\overline{\nu_{j_m}}\;:\;E\;\overline{\xi_{j_m}})\;=\;b_{0,j_m,k_0}\;\rangle^{n_m,p_0}_{j_m=1,k_0=1};\\
&\quad\ldots\;;\\
&\quad\langle\, f_{m-1,k_{m-1}}\;(\;\mathcal{E}_{m,j_m}\;\overline{\nu_{j_m}}\;:E\;\overline{\xi_{j_m}})\;=\;b_{m-1,j_m,k_{m-1}}\;\rangle^{n_m,p_{m-1}}_{j_m=1,k_{m-1}=1}\;]\!]\;=\\
&\langle\;\mathbf{instance}\;(\;S\;(\overline{D}^{m}\delta_m\;(\mathbf{E}\;(\overline{D}^{m}\delta_m)))\\
&\qquad,\;S\;(\overline{D}^{m}\delta_m\;(E_0\;(\overline{D}^{m}\delta_m)))\\
&\qquad,\;\ldots\\
&\qquad,\;S\;(\overline{D}^{m}\delta_m\;(E_m\;(\overline{D}^{m}\delta_m)\;))\\
&\qquad)\;\Rightarrow\;F_i\;(\overline{D}^{m}_{i}\delta_m)\;(E_m\;(\overline{D}^{m}\delta_m))\;\mathbf{where}\\
&\quad\langle\, f_{i,k_i}\;(\_ : P\;(\overline{D}^{m}\delta_m))\\
&\qquad(\;\mathcal{E}_{m,j_m}\;\overline{\nu_{j_m}}\;:\;E_m\;(\overline{D}^{m}\delta_m)\;\overline{\xi_{j_m}})\;=\;\mathcal{T}^{e}_{method}[\![b_{i,j_m,k_i}]\!]\;\rangle^{n_m,p_i}_{j_m=1,k_i=1}\\
&\rangle^{m-1}_{i=0}
\end{aligned}
$$

Figure 10c. Translation for new equations on existing functions in the m_{th} extension module.

$$\mathcal{T}^{data}\llbracket\, \textbf{extend data}\; E\,\overline{\alpha} \;=\; \langle\; \mathcal{E}_{m,j}\;\overline{\tau_j}\;\rangle_{j_m=1}^{n_m}\;;$$
$$\quad \langle\, f_{m,k_m} \;:\; \sigma_{m,k_m}\;\rangle_{k_m=1}^{p_m};\; \langle\, f_{m,k_m}\;(\,\mathcal{E}_{m,j_m}\;\overline{\nu_{m,j_m}}\;:\;E\;\overline{\xi_{m,j_m}})\;=\;b_{m,j_m,k_m}\;\rangle_{j_m=1,k_m=1}^{n_m,p_m}\;\rrbracket\;=$$

The m_{th} extension component type

$$\textbf{data}\; E_m\;(\delta\;:\;\mathcal{T}_{kind}^{\overline{\alpha}}\llbracket\overline{\alpha}\rrbracket)\;\overline{\alpha}\;=\;\langle\;\mathcal{E}_{m,j_m}\;\overline{\mathcal{T}_E^{\tau}}\llbracket\overline{\tau_j}\rrbracket\;\rangle_{j_m=1}^{n_m};$$

The m_{th} functionality class

$$\textbf{class}\;(S\;(\overline{D}^m\delta_m\;\beta),\;F_{m-1}\;(D_m\;\delta_m)\;\beta)\;\Rightarrow\;F_m\;\delta_m\;\beta\;\textbf{where}$$
$$\quad\langle\, f_{m,k_m}\;:\;\mathcal{T}_{method}^{\sigma}\llbracket\sigma_{m,k_m}\rrbracket\;\rangle_{k_m=1}^{p_m};$$

The m_{th} explicit dictionary

$$\textbf{data}\;D_m\;\delta_m\;\beta\;=\;\mathcal{D}_m\;\{\;\langle\,f'_{m,k_m}\;:\;\mathcal{T}_{method}^{\sigma}\llbracket\sigma_{m,k_m}\rrbracket;\;\rangle_{k_m=1}^{p_m}\;;$$
$$\qquad\qquad ,\;d_{m+1}\;:\;(\delta_m\;\beta)\;\}$$

Unwrapping instance

$$\textbf{instance}\;S\;(\overline{D}^m\delta_m\;(\mathbf{E}\;(\overline{D}^m\delta_m)))\;\Rightarrow\;F_m\;\delta_m\;(\mathbf{E}\;(\overline{D}^m\delta_m))\;\textbf{where}$$
$$\quad\langle\,f_{m,k_m}\;(\_:P\;(\overline{D}^m\delta_m))\;(\,\mathcal{E}\;x\;:\;\mathbf{E}\;(\overline{D}^m\delta_m)\;\overline{\xi})\;=$$
$$\qquad\mathcal{T}_{unwrap}^{\sigma}(f_{m,k_m},\;x,\;P\;(\overline{D}^m\delta_m))\llbracket\sigma_{m,k_m}\rrbracket\;\rangle_{k_m=1}^{p_m}$$

Functionality instances (for component types $0 \leq i \leq m$)

$$\langle\;\textbf{instance}\;(\;S\;(\overline{D}^m\delta_m\;(\mathbf{E}\;(\overline{D}^m\delta_m)\;))$$
$$\qquad ,\;S\;(\overline{D}^m\delta_m\;(E_0\;(\overline{D}^m\delta_m)\;))$$
$$\qquad ,\;\ldots$$
$$\qquad ,\;S\;(\overline{D}^m\delta_m\;(E_m\;(\overline{D}^m\delta_m)\;))$$
$$\qquad)\;\Rightarrow\;F_m\;\delta_m\;(E_i\;(\overline{D}^m\delta_m)\;)\;\textbf{where}$$
$$\quad\langle\,f_{m,k_m}\;(\_:P\;(\overline{D}^m\delta_m))\;(\,\mathcal{E}_{i,j_i}\;\overline{\nu_{j_i}}\;:\;E_i\;(\overline{D}^m\delta_m)\;\overline{\xi_{j_i}})\;=\;\mathcal{T}_{method}^{e}\llbracket b_{i,j_i,k_m}\rrbracket\;\rangle_{j_i=1,k_m=1}^{n_i,p_m}$$
$$\rangle_{i=0}^{m}$$

Capping class, type and instances

$$\textbf{data}\;\hat{D}_m\;\beta\;;$$
$$\textbf{class}\;F_m\;\hat{D}_m\;\beta\;\Rightarrow\;\hat{F}_m\;\beta\;;$$
$$\textbf{instance}\;\hat{F}_m\;(\mathbf{E}\;(\overline{D}^m\;\hat{D}_m))\;;$$
$$\langle\;\textbf{instance}\;\hat{F}_m\;(E_i\;(\overline{D}^m\;\hat{D}_m))\;;\;\rangle_{i=0}^{m}$$

"Knot tying" instance

$$\textbf{instance}\;\hat{F}_m\;\beta\;\Rightarrow\;S\;(\overline{D}^m\;\hat{D}_m\;\beta)\;\textbf{where}$$
$$\quad d_1\;=\;D_1\;\{\;\langle\,f'_{1,k}\;=\;f_{1,k}\;\rangle_{k=1}^{p_1}$$
$$\qquad\qquad d_2\;=\;D_2\;\{\;\langle\,f'_{2,k}\;=\;f_{2,k},\;\rangle_{k=1}^{p_2}$$
$$\qquad\qquad\qquad\cdots$$
$$\qquad\qquad\qquad\ldots\;d_m\;=\;D_m\;\{\;\langle\,f'_{m,k}\;=\;f_{m,k}\;\rangle_{k=1}^{p_m},\;d_{m+1}\;=\;\bot\}\;\ldots\}$$

Smart constructors

$$\langle\;\varepsilon_{m,j_m}:\mathcal{T}_{smart}^{\overline{\tau}}\llbracket\overline{\tau_{j_m}}\rrbracket;\;\varepsilon_{m,j_m}\;(\_:P\;\delta)\;=\;\mathcal{T}_{smart}^{e}(m,j_m)\llbracket\overline{\tau_{j_m}}\rrbracket\;\rangle_{j_m=1}^{n_m}$$

Figure 10d. Translation for *extend data* declaration and new function for the m_{th} extension module.

$$\mathcal{T}^{\overline{\alpha}}_{kind}[\![\alpha_1,\ \ldots,\ \alpha_k]\!] \quad = \overbrace{(\star\ \to\ \ldots\ \to\ \star)}^{k+1} \to \star$$

$$\begin{array}{ll}
\mathcal{T}^{\sigma}_{method}[\![\forall\overline{\alpha}.E\ \overline{\tau}\ \to\ \xi]\!] & = \forall\overline{\alpha}.\ P(\overline{D}^m\delta_m)\ \to\ \beta\ \overline{\tau}\ \to\ \mathcal{T}^{\tau}_{method}[\![\xi]\!] \\
\mathcal{T}^{\tau}_{method}[\![\alpha]\!] & = \alpha \\
\mathcal{T}^{\tau}_{method}[\![T]\!] & = \left\{ \begin{array}{ll} T\ (\overline{D}^m\delta_m) & \text{, if } T \in \Gamma(E) \\ T & \text{, otherwise} \end{array} \right. \\
\mathcal{T}^{\tau}_{method}[\![E]\!] & = E\ (\overline{D}^m\delta_m) \\
\mathcal{T}^{\tau}_{method}[\![\tau_1\ \tau_2]\!] & = \mathcal{T}^{\tau}_{method}[\![\tau_1]\!]\ \mathcal{T}^{\tau}_{method}[\![\tau_2]\!]
\end{array}$$

$$\begin{array}{ll}
\mathcal{T}^{e}_{method}[\![f_{i,k_i}]\!] & = \left\{ \begin{array}{ll} f'_{i,k_i}\ \mathcal{T}^{dict}(i)\ (\bot : P\ (\overline{D}^m\delta_m)) & \text{, if } i > 0 \\ f_{i,k_i}\ (\bot : P\ (\overline{D}^m\delta_m)) & \text{, otherwise} \end{array} \right. \\
\mathcal{T}^{e}_{method}[\![x]\!] & = \left\{ \begin{array}{ll} x\ (\bot : P\ (\overline{D}^m\delta_m)) & \text{, if } x \in \Delta(E) \\ x & \text{, otherwise} \end{array} \right. \\
\mathcal{T}^{e}_{method}[\![\lambda x\ :\ \tau.e]\!] & = \lambda x\ :\ \mathcal{T}^{\tau}_{method}[\![\tau]\!].\ \mathcal{T}^{e}_{method}[\![e]\!] \\
\mathcal{T}^{e}_{method}[\![\ \mathcal{E}_{i,j_i}]\!] & = \varepsilon_{i,j_i}\ (\bot :\ P\ (\overline{D}^m\delta_m)) \\
\mathcal{T}^{e}_{method}[\![\ \mathcal{C}]\!] & = \mathcal{C} \\
\mathcal{T}^{e}_{method}[\![e_1\ e_2]\!] & = \mathcal{T}^{e}_{method}[\![e_1]\!]\mathcal{T}^{e}_{method}[\![e_2]\!]
\end{array}$$

$$\mathcal{T}^{\sigma}_{unwrap}(f_{i,k_i}, x, \gamma)[\![\forall\overline{\alpha}.\ E\ \overline{\tau}\ \to\ \tau_1\ \to\ \ldots\ \to\ \tau_n\ \to\ \tau_{n+1}]\!]\ =$$
$$\left\{ \begin{array}{ll}
\lambda x_1 : \mathcal{T}^{\tau}_{method}[\![\tau_1]\!] & \\
\ldots & \\
\lambda x_n : \mathcal{T}^{\tau}_{method}[\![\tau_n]\!].f_0\ (\bot : \gamma)\ x\ x_1\ \ldots\ x_n & \text{, if } i = 0 \\
\lambda x_1 : \mathcal{T}^{\tau}_{method}[\![\tau_1]\!] & \\
\ldots & \\
\lambda x_n : \mathcal{T}^{\tau}_{method}[\![\tau_n]\!].f_{i,k_i}\ \mathcal{T}^{dict}(i)\ (\bot : \gamma)\ x\ x_1\ \ldots\ x_n & \text{, otherwise}
\end{array} \right.$$

Figure 10e. Translation rules

$$\begin{array}{ll}
\mathcal{T}^{\overline{\tau}}_{smart}[\![\overline{\tau}]\!] & = \forall\ \delta_m.\forall\ \overline{\alpha}.\ (\ S\ (\overline{D}^m\delta_m\ (\mathbf{E}\ (\overline{D}^m\delta_m))) \\
& \qquad ,\ S\ (\overline{D}^m\delta_m\ (E_0\ (\overline{D}^m\delta_m))) \\
& \qquad ,\ \ldots \\
& \qquad ,\ S\ (\overline{D}^m\delta_m\ (E_m\ (\overline{D}^m\delta_m))))\ \Rightarrow \\
& \quad connect\ (P\ (\overline{D}^m\delta_m),\ \overline{\mathcal{T}^{\tau}_{method}}\ [\![\ \overline{\tau}\ ]\!],\ \mathbf{E}\ (\overline{D}^m\delta_m)\ \overline{\alpha}) \\
\mathcal{T}^{\tau}_{smart}[\![T]\!] & = \left\{ \begin{array}{ll} T\ (\overline{D}^m\delta_m) & \text{, if } T \in \Gamma(E) \\ T & \text{, otherwise} \end{array} \right. \\
\mathcal{T}^{\tau}_{smart}[\![E]\!] & = \mathbf{E}\ (\overline{D}^m\delta_m) \\
\mathcal{T}^{\tau}_{smart}[\![\tau_1\ \tau_2]\!] & = \mathcal{T}^{\tau}_{smart}[\![\tau_1]\!]\ \mathcal{T}^{\tau}_{smart}[\![\tau_2]\!]
\end{array}$$

$$connect\ [\![\ \tau_1,\ \ldots,\ \tau_k\ ]\!] \ = \tau_1\ \to\ \cdots \to\ \tau_k$$

$$\mathcal{T}^{e}_{smart}(i,j)[\![\tau_1,\ \ldots,\ \tau_k]\!] = \lambda x_1 : \tau_1 \ldots \lambda x_k : \tau_k.\ \mathcal{E}\ (\ \mathcal{E}_{i,j}\ x_1\ \ldots\ x_k : E_{i,j}\ (\overline{D}^i\delta_i)\ \overline{\alpha})$$

$$\begin{array}{ll}
\mathcal{T}^{dict}(i) & = (d_i\ (\ldots\ (d_2\ d_1)\ldots)) \\
\mathcal{T}^{\tau}_{E}[\![\alpha]\!] & = \alpha \\
\mathcal{T}^{\tau}_{E}[\![E]\!] & = \mathbf{E}\ \delta \\
\mathcal{T}^{\tau}_{E}[\![T]\!] & = \left\{ \begin{array}{ll} T\ (\overline{D}^m\delta_m) & \text{, if } T \in \Gamma(E) \\ T & \text{, otherwise} \end{array} \right. \\
\mathcal{T}^{\tau}_{E}[\![\tau_1\ \tau_2]\!] & = \mathcal{T}^{\tau}_{E}[\![\tau_1]\!]\ \mathcal{T}^{\tau}_{E}[\![\tau_2]\!]
\end{array}$$

Figure 10f. More translation rules

B. Dictionary translation of module *F0_Alpha*

The code demonstrates the resulting of performing dictionary translation (in the style of Wadler and Blott [14]) on the code in Figures 4a – 4g. It makes it clear where the recursive dictionaries are built.

```
module Alpha
where
data P d
u = ⊥
 {-DI stands for dictionary implicity. D is an explicit dictionary -}
 {-class Sat a where dict :: a -}
data SatDI a = SatDI{ dict :: a }
data Exp (cxt :: * → *) = forall b. MkExp (AlphaDI cxt b, b)
data Exp_0 cxt = Var String
   | Lam String (Exp cxt)
   | App (Exp cxt) (Exp cxt)
 {-class Sat (cxt b) ⇒ Alpha cxt b where alpha :: P cxt → b → (String, String) → Exp cxt -}
data AlphaDI cxt b = AlphaDI{ alpha :: P cxt → b → (String, String) → Exp cxt }

 {-instance (Sat (cxt (Exp cxt)), Sat (cxt (Exp_0 cxt))) ⇒ Alpha cxt (Exp_0 cxt) -}
alphaDExp0 :: forall cxt. (SatDI (cxt (Exp cxt)), SatDI (cxt (Exp_0 cxt))) →
  AlphaDI cxt (Exp_0 cxt)
alphaDExp0 (satExp, satExp0) = AlphaDI{ alpha = alpha′ }
  where
    alpha′ (_ :: P cxt) (Var v) =
      λs → var (satExp, satExp0) (u :: P cxt) (swap s v)
    alpha′ (_ :: P cxt) (Lam v body) =
      case body of
        MkExp (alphaD, body′) →
          λs → lam (satExp, satExp0) (u :: P cxt) (swap s v)
              (alpha alphaD (u :: P cxt) body′ s)
    alpha′ (_ :: P cxt) (App a b) =
      case a of
        MkExp (alphaDa, a′) →
          case b of
            MkExp (alphaDb, b′) → λs →
              app (satExp, satExp0)
                (u :: P cxt) (alpha alphaDa (u :: P cxt) a′ s)
                  (alpha alphaDb (u :: P cxt) b′ s)

 {-instance Sat (cxt (Exp cxt)) ⇒ Alpha cxt (Exp cxt) -}
alphaDExp :: forall cxt. SatDI (cxt (Exp cxt)) → AlphaDI cxt (Exp cxt)
alphaDExp satExp = AlphaDI{ alpha = alpha′ }
  where
    alpha′ (_ :: P cxt) exp =
      case exp of
        MkExp (alphaD, e) →
          λs → alpha alphaD (u :: P cxt) e s
swap :: (String, String) → String → String
swap ((a, b) :: (String, String)) = λ(o :: String) → if a == o then b else o
```

$var :: forall\ cxt. (SatDI\ (cxt\ (Exp\ cxt))$
$\quad , SatDI\ (cxt\ (Exp\_0\ cxt))) \rightarrow P\ cxt \rightarrow String \rightarrow Exp\ cxt$
$var\ (satExp, satExp0)\ (\_ :: P\ cxt) =$
$\quad \lambda(x1 :: String) \rightarrow MkExp\ (alphaDExp0\ (satExp, satExp0), Var\ x1)$
$lam :: forall\ cxt. (SatDI\ (cxt\ (Exp\ cxt))$
$\quad , SatDI\ (cxt\ (Exp\_0\ cxt))) \rightarrow P\ cxt \rightarrow String \rightarrow Exp\ cxt \rightarrow Exp\ cxt$
$lam\ (satExp, satExp0)\ (\_ :: P\ cxt) =$
$\quad \lambda(x1 :: String)\ (x2 :: Exp\ cxt) \rightarrow MkExp\ (alphaDExp0\ (satExp, satExp0), Lam\ x1\ x2)$
$app :: forall\ cxt. (SatDI\ (cxt\ (Exp\ cxt))$
$\quad , SatDI\ (cxt\ (Exp\_0\ cxt))) \rightarrow P\ cxt \rightarrow Exp\ cxt \rightarrow Exp\ cxt \rightarrow Exp\ cxt$
$app\ (satExp, satExp0)\ (\_ :: P\ cxt) =$
$\quad \lambda(x1 :: Exp\ cxt)\ (x2 :: Exp\ cxt) \rightarrow MkExp\ (alphaDExp0\ (satExp, satExp0), App\ x1\ x2)$

--
-- Capping class
--

data $AlphaEnd\ b$

{-class Alpha AlphaEnd b ⇒ AlphaCap b -}

data $AlphaCapDI\ b = AlphaCapDI\{alphaD :: AlphaDI\ AlphaEnd\ b\}$

--
-- d and d0 are the recursive dictionaries for ”instance AlphaCap (Exp AlphaEnd)” and
-- ”instance AlphaCap (Exp0 AlphaEnd)” respectively.
--

{-instance AlphaCap (Exp0 AlphaEnd) -}

$d0 :: AlphaCapDI\ (Exp\_0\ AlphaEnd)$
$d0 = AlphaCapDI\{alphaD = alphaDExp0\ (satD\ d, satD\ d0)\}$

{-instance AlphaCap (Exp AlphaEnd) -}

$d :: AlphaCapDI\ (Exp\ AlphaEnd)$
$d = AlphaCapDI\{alphaD = alphaDExp\ (satD\ d)\}$

{-instance AlphaCap b =¿ Sat (AlphaEnd b) -}

$satD :: AlphaCapDI\ b \rightarrow SatDI\ (AlphaEnd\ b)$
$satD\ \_ = SatDI\{dict = error$ `"Capped at Alpha"`$\}$

{-test = alpha (var (u::P AlphaEnd) ”x”) (”x”, ”y”) -}

$test =$ **let** $p = u :: P\ AlphaEnd$
in $alpha\ (alphaDExp\ (satD\ d))\ p$
$\quad (var\ (satD\ d, satD\ d0)\ p$ `"x"`$)$ $($`"x"`, `"y"`$)$