Open Data Types and Open Functions

Andres Löh Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
{loeh,ralf}@informatik.uni-bonn.de

Abstract

The problem of supporting the modular extensibility of both data and functions in one programming language at the same time is known as the expression problem. Functional languages traditionally make it easy to add new functions, but extending data (adding new data constructors) requires modifying existing code. We present a semantically and syntactically lightweight variant of open data types and open functions as a solution to the expression problem in the Haskell language. Constructors of open data types and equations of open functions may appear scattered throughout a program with several modules. The intended semantics is as follows: the program should behave as if the data types and functions were closed, defined in one place. The order of function equations is determined by *best-fit* pattern matching, where a specific pattern takes precedence over an unspecific one. We show that our solution is applicable to the expression problem, generic programming, and exceptions. We sketch two implementations: a direct implementation of the semantics, and a scheme based on mutually recursive modules that permits separate compilation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords expression problem, extensible data types, extensible functions, functional programming, Haskell, generic programming, extensible exceptions, mutually recursive modules

1. Introduction

Suppose we have a very simple language of expressions, which consists only of integer constants:

data $Expr = Num\ Int$

We intend to grow the language, and add new features as the need for them arises. Here is an interpreter for expressions:

$$eval :: Expr \to Int$$
$$eval\ (Num\ n) = n$$

There are two possible directions to extend the program: first, we can add new functions, such as a conversion function from expressions to strings; second, we can add new forms of expressions by adding new constructors.

PPDP'06 July 10–12, 2006, Venice, Italy.

Preferably, such extensions should be possible without modifying the previously written code, because we want to organize different aspects separately, and the existing code may have been written by a different person and exist in a closed library.

Alas, Haskell natively supports only one of the two directions of extension. We can easily add a new function, such as

$$toString :: Expr \to String$$
$$toString\ (Num\ n) = show\ n$$

but we cannot grow our language of expressions: in Haskell, all constructors of the *Expr* data type must be defined the very moment when *Expr* is introduced. We therefore cannot add new constructors to the *Expr* data type without touching the original definition.

The above problem of supporting the modular extensibility of both data and functions in one programming language at the same time has been formulated by Wadler [1] and is called the *expression problem*. As we have just witnessed, functional languages make it easy to add new functions, but extending data (adding new data constructors) requires modifying existing code. In object-oriented languages, extension of data is directly supported by defining new classes, but adding new functions to work on that data requires changing the class definitions. Using the Visitor design pattern, one can simulate the situation of functional programming in an object-oriented language: one gains extensibility for functions, but loses extensibility of data at the same time.

Many partial and proper solutions to the expression problem have been proposed over the years. However, much of this work suffers from one or more of the following disadvantages:

- It is focused on object-oriented programming, and cannot directly be translated to a functional language.
- It introduces rather complex extensions to the type system, such as multi-methods [2, 3] or mixins [4].
- Open entities have their own special syntax or severe limitations, thereby forcing a programmer to decide in advance whether an entity should be open or not, because there is no easy way to switch later.

Closest to our proposal is the work of Millstein *et al.* [3], who describe the addition of hierarchical classes and extensible functions to ML. A discussion of this and other related work can be found in Section 6.

In this paper, we present a semantically and syntactically lightweight variant of *open data types* and *open functions* as a solution to the expression problem in the functional programming language Haskell. By declaring *Expr* (from the example above) as an open data type, we can add new constructors at any time and at any point in the program. If we want to evaluate or print the newly introduced expression forms, we have to adapt *eval* and *toString*. If these functions are declared as open functions, we can add new

defining equations for the functions at a later point in the program. The intended semantics is as follows: the program should behave as if the data types and functions were closed, defined in one place.

The main design goal of our proposal is simplicity: open data types and open functions are easy to add to the language, and their semantics is simple to explain as a source-to-source transformation. In particular, the type system and the operational semantics of the original language are unaffected by the addition. Interestingly, we can implement our proposal in such a way that it supports separate compilation: we employ Haskell's support for mutually recursive modules to tie the recursive knot of open functions (see Section 5.2).

In addition to the expression problem, we present two other applications of open data types and open functions: generic programming and extensible exceptions.

The rest of the paper is structured as follows: In Section 2, we present the syntax and informal semantics of open data types and open functions. We use the expression problem as a running example and show how it is solved using the new features. We then look at other applications (Section 3). A formal semantics is given in Section 4. We investigate possible implementations in Section 5. We then discuss related work (Section 6), before we conclude in Section 7.

2. Open data types and open functions

In this section, we describe how to declare open data types and open functions, and how to add new constructors to open data types and new equations to open functions. We also explain the semantics of these constructs, and we discuss the problem of pattern matching for open functions.

We strive for a simple and pragmatic solution to the problem of open data types. Therefore, we draw inspiration from a well-known and widely used Haskell feature that *is* extensible: type classes. Instances for type classes can be defined at any point in the program, but type classes also have a number of restrictions that make this possible: Type classes have a global, program-wide meaning. Type class and instance declarations may only appear at the top level of a module. All instances are valid for the entire program, and instance declarations cannot be hidden.

We use a similar approach for open data types and functions: they have a global, program-wide meaning. Open entities can only be defined and extended at the top level of a module. All constructors and function equations are valid for the entire program, and cannot be hidden.

2.1 Open data types

The following declaration introduces an open data type of expressions:

$$\textbf{open data}\ \mathit{Expr} :: *$$

The keyword **open** flags *Expr* as open. In contrast to ordinary data type declarations, the declaration does not list any constructors. Instead, constructors of an open data type are separate entities that can be declared by giving a type signature for them at any point in the program, for example:

$$\mathit{Num} :: \mathit{Int} \to \mathit{Expr}$$

The fact that we are giving a type signature for a constructor identifier (an identifier starting with an uppercase letter or a colon) signals that this is a constructor of an open data type. The result type *Expr* determines the data type that we are extending.

2.2 Open functions

Open functions are introduced by a mandatory type signature, also marked with the **open** keyword:

$$\textbf{open}\ \mathit{eval} :: \mathit{Expr} \to \mathit{Int}$$

Note that only top-level functions can be marked as open. The defining equations for an open function can be provided at any point of the program where the name of the function is in scope, using the normal syntax for function definitions:

$$\mathit{eval}\ (\mathit{Num}\ n) = n$$

If we leave it at that, we have a program consisting of an *Expr* data type with only one constructor, and an *eval* function consisting of only one equation, just like in the introduction.

2.3 The expression problem

Open data types and open functions support both directions of extension. A new function can be added as usual, except that we also mark it as open:

$$\begin{aligned}&\textbf{open}\ \mathit{toString} :: \mathit{Expr} \to \mathit{String}\\ &\mathit{toString}\ (\mathit{Num}\ n) = \mathit{show}\ n\end{aligned}$$

It is now easy to add a new constructor for *Expr*. All we have to do is provide its type signature:

$$\mathit{Plus} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}$$

Of course, if we do not adapt *eval* and *toString* and call it on expressions constructed with *Plus*, we get a pattern match failure. However, since *eval* and *toString* are open functions, we can just provide the missing equations:

$$\begin{aligned}&\mathit{eval}\quad (\mathit{Plus}\ e_1\ e_2) = \mathit{eval}\ e_1 + \mathit{eval}\ e_2\\ &\mathit{toString}\ (\mathit{Plus}\ e_1\ e_2) = \texttt{"("} +\!\!+ \mathit{toString}\ e_1 +\!\!+ \texttt{"+"}\\ &\qquad\qquad\qquad\qquad +\!\!+ \mathit{toString}\ e_2 +\!\!+ \texttt{")"}\end{aligned}$$

Note that we have extended our program without modifying existing code.

2.4 Semantics

Let us now turn to the semantics of open data types and open functions. The driving goal of our proposal is to be as simple as possible: a program should behave as if all open data types and open functions were closed, defined in one place. Apart from the syntactic peculiarity that they are defined in several places, open data types and open functions define ordinary data types and functions. Their addition to the language affects neither the type system nor the operational semantics of the underlying language.

To return to our example: the meaning of the program written above is exactly the same as the meaning of the following program without open data types or open functions:

$$\begin{aligned}&\textbf{data}\ \mathit{Expr} = \mathit{Num}\ \mathit{Int}\\ &\qquad\quad \mid \mathit{Plus}\ \mathit{Expr}\ \mathit{Expr}\\ &\mathit{eval} :: \mathit{Expr} \to \mathit{Int}\\ &\mathit{eval}\ (\mathit{Num}\ n) \quad = n\\ &\mathit{eval}\ (\mathit{Plus}\ e_1\ e_2) = \mathit{eval}\ e_1 + \mathit{eval}\ e_2\\ &\mathit{toString} :: \mathit{Expr} \to \mathit{String}\\ &\mathit{toString}\ (\mathit{Num}\ n) \quad = \mathit{show}\ n\\ &\mathit{toString}\ (\mathit{Plus}\ e_1\ e_2) = \texttt{"("} +\!\!+ \mathit{toString}\ e_1 +\!\!+ \texttt{"+"}\\ &\qquad\qquad\qquad\qquad +\!\!+ \mathit{toString}\ e_2 +\!\!+ \texttt{")"}\end{aligned}$$

Compared to the original program, all the constructors are collected into a single definition of *Expr*, and all function equations are collected to provide a single definition of *eval* and *toString*. In particular, the recursive calls in the *Plus*-equations of *eval* and *toString* point to the complete definitions of *eval* and *toString*.

Let us consider the order in which the previously separate declarations are grouped. The order of data type constructors is mostly

irrelevant. The only exception occurs when applying the **deriving** construct to an open data type (see Section 4.4).

However, the order in which the defining equations of a function appear is significant. Haskell employs *first-fit* pattern matching: the equations of functions are matched linearly, from top to bottom. The first equation that fits is selected.

2.5 Best-fit left-to-right pattern matching

For open functions, first-fit pattern matching is not suitable. To see why, suppose that we want to provide a default definition for *toString* in order to prevent pattern matching failures, stating that everything without a specific definition is ignored in the string representation:

```
toString _ = ""
```

Using first-fit pattern matching, this equation effectively closes the definition of *toString*. Later equations cannot be reached at all. Furthermore, if equations of the function definition are scattered across multiple modules, it is unclear (or at least hard to track) in which order they will be matched.

We therefore adopt a different scheme for open functions, called *best-fit left-to-right* pattern matching. We describe the details in Section 4.3, but the idea is that the most specific match (rather than the first match) wins. This makes the order in which equations of the function appear irrelevant. In the example above, it ensures that the default case for *toString* will be chosen only if no other equation matches. To be precise, the definition of *toString* with a default case will behave as in the following standard Haskell definition, in which the default case occurs last:

```
toString :: Expr → String
toString (Num n)      = show n
toString (Plus e1 e2) = "(" ++ toString e1 ++ "+"
                            ++ toString e2 ++ ")"
toString _            = ""
```

Before we describe the details and corner cases of the semantics in Section 4, let us investigate two application areas of open datatypes and open functions.

3. Applications

In this section, we present two applications where open data types and open functions help to overcome severe limitations. First, we show how lightweight generic programming becomes modular with the introduction of an extensible type of type representations. Second, we describe a replacement for Haskell's ad-hoc approach to exceptions.

3.1 Generic programming

A generic function is a function that is defined once, but works on many data types, by viewing all data types in a uniform way. Classical examples of generic functions are equality and comparison functions, parsing and pretty-printing, traversals over large data structures such as abstract syntax trees etc.

Many ways to incorporate generic programming into functional programming languages have been proposed, ranging from lightweight libraries [5] to full-fledged language extensions [6]. One particularly attractive approach, recently discussed in a flurry of papers [7, 8, 9], is to base generic functions on *overloaded* functions, i.e., functions that are parametrized by a type representation.

A type of type representations is easy to define using a *generalized algebraic data type* (GADT), and to make it more useful, we mark it as open:

```
open data Type :: * → *
Int    :: Type Int
Char   :: Type Char
Pair   :: Type a → Type b → Type (a,b)
```

A value of type *Type a* is a representation of type *a*. Using pattern matching on such a representation, we can group specialized functions for different data types in a single definition. As an example, consider an overloaded function that turns a value into its string representation:

```
open toString :: Type a → a → String
toString Int      n      = show n
toString Char     c      = show c
toString (Pair a b) (x,y) =
  "(" ++ toString a x ++ "," ++ toString b y ++ ")"
```

Using a slightly more involved definition, we can produce a representation with a minimal amount of parentheses such as Haskell's built-in function *show*, or a pretty-printer that handles indentation and alignment.

Whenever we add new data types, we must equip existing overloaded and generic functions with new equations for the new data type. That is why the *Type* data type and overloaded functions are open.

If we add a new type, say binary trees

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

we adapt *Type* and overloaded functions accordingly:

```
Tree :: Type a → Type (Tree a)
toString (Tree a) Empty        = "Empty"
toString (Tree a) (Node l x r) =
  "(Node " ++ toString (Tree a) l ++ " "
           ++ toString a x ++ " "
           ++ toString (Tree a) r ++ ")"
```

Generic functions are built on top of overloaded functions and reduce the overhead of adding a new data type. Only a single overloaded function, which allows to access the structure of all data types in a uniform way, must be augmented with a new equation for the new type; all the generic functions then work on the new type without modification. However, the extensibility of this single overloaded function and the extensibility of the data type *Type* are still required.

The approach to generic programming using type representations is particularly attractive, because overloaded and generic functions are ordinary Haskell functions, and therefore first-class citizens. Generic functions can be passed as arguments to other functions, or returned as results of other functions. Pattern matching on type arguments boils down to pattern matching on values. However, without open data types, such an approach is only useful for experimentation or in a closed setting where all types are known in advance. In general, a generic programming library must be able to deal with the case that new, application-specific data types are defined later by a user. With open data types, this is possible, and a lightweight generic programming framework can be distributed as a library.

Furthermore, this example indicates that there is no problem in having open GADTs. For GADTs, the types of the constructors are less restricted, but the result type of a constructor still uniquely determines the data type it belongs to.

3.2 Exception handling

Haskell (or, more precisely, GHC) has the following exception interface:

```
throw :: Exception → a
catch :: IO a → (Exception → IO a) → IO a
```

program	*program*	$::= module^*$
module	*module*	$::=$ **module** *moduleid* **where** $\{decl^*\}$
declaration	*decl*	$::=$ **import** *moduleid*
		$\mid$ **data** *dataid* $::$ *kind* **where** $\{consig^*\}$
		$\mid$ **open data** *dataid* $::$ *kind*
		$\mid$ *consig*
		$\mid$ *varid* $::$ *type*
		$\mid$ **open** *varid* $::$ *type*
		$\mid$ *equation*
function equation	*equation*	$::=$ *varid* $pat^* =$ *expr*
constructor type signature	*consig*	$::=$ *conid* $::$ *type*
pattern	*pat*	$::=$ *varid* $\mid$ (*conid* $varid^*$)
kind	*kind*	$::= * \mid (kind \to kind)$
type	*type*	$::=$ *varid* $\mid$ *conid* $\mid$ (*type type*) $\mid$ $(type \to type)$
expression	*expr*	$::=$ *varid* $\mid$ *conid* $\mid$ (*expr expr*)

Figure 1. Syntax of the core language for open data types and open functions

An exception can be thrown anywhere using *throw*, but only caught in a computation using *catch*. The first argument to *catch* is the computation that is watched for exceptions, the second argument is the *exception handler*: if the computation raises an exception, the handler is called with the exception as an argument.

Different applications require different kinds of exceptions. For example, an exception caused due to a filesystem permission problem should additionally provide the file name that could not be accessed, and the permissions of that file. An array exception should provide the index that was determined to be out of bounds.

If we look at the definition of the type *Exception* in the GHC libraries, we learn that there are several predefined constructors for frequent errors. If an application-specific error arises, such as when an illegal key is passed to the lookup function of a finite map library, we must therefore try to find a close match among the predefined constructors, where we would prefer to define a new tailor-made constructor *KeyNotFound*.

The problem of extensible exceptions is so pressing that the ML family of languages has a special language construct for exceptions. In OCaml one can define

exception *KeyNotFound* **of** *key*;;

to extend the exception language. The **exception** construct declares a new constructor *KeyNotFound* for the built-in expression data type *exn*. The constructor is parametrized over an argument of type *key*.

An exception in OCaml can be thrown using the function

$raise : exn \to {}'a$

which in OCaml syntax means that *raise* takes an exception *exn* to any type $'a$. If the lookup function does not find a key in a given finite map, it can trigger an exception:

$lookup\ k\ fm = \ldots raise\ (KeyNotFound\ k) \ldots$

Associated with the **exception** keyword is a special form of **case** statement to catch exceptions, called **try**:

try ... **with**
  | *KeyNotFound k* $\to$ ...

If none of the patterns in a **try** statement match, the exception is automatically propagated (i.e., re-raised).

With open data types, we can model the same approach in Haskell. We declare the data type of exceptions to be open:

open data *Exception* $:: *$

Declaring a new exception boils down to declaring a new constructor of the exception type

$KeyNotFound :: Key \to Exception$

We can throw the exception using

$lookup\ k\ fm = \ldots throw\ (KeyNotFound\ k) \ldots$

and catch it using a combination of *catch* and a normal **case** statement:

```
catch (...)
  (λe → case e of KeyNotFound k → ...
                  _             → return (throw e))
```

The difference to the OCaml version is that we need an explicit default case for the handler, in which we explicitly re-raise the exception. Without this default case, we would get a run-time error if none of the provided handlers matches.

Interestingly, we have no need for open functions in the context of exception handling. Handlers are typically written as local functions that match against a few known kinds of exceptions. Unknown exceptions are always propagated using a default case, rendering extension of the handler at a later time unnecessary.

We have shown that open data types subsume ML-style exception handling. A special-purpose language construct for exceptions is not required. While it is also possible to implement extensible exceptions in Haskell using dynamic typing via the *Typeable* type class, we believe that our variant of exception handling is more in the spirit of typed functional programming languages.

4. Semantics

After we have seen the proposed language extension "in action", let us now discuss the precise semantics.

4.1 Core language

To abstract from the unnecessary ballast that Haskell as a full-blown programming language carries around, we will use a small core language containing only the relevant features, with the syntax given in Figure 1. The syntax is shown again, as Haskell data types, in Figure 2, so that we can use Haskell code to specify parts of the semantics concisely.

A program consists of several modules. A module has a name and contains declarations. We only consider a few forms of declarations. Other modules can be imported (but there are no export or import lists). We can define data types and open data types. We can

```
data Program     = Program [Module]
data Module      = Module ModuleId [Declaration]
data Declaration = ImportDecl ModuleId
                 | DataDecl DataId Kind [ConSig]
                 | OpenDataDecl DataId Kind
                 | Constructor ConSig
                 | TypeSig VarId Type
                 | OpenTypeSig VarId Type
                 | Function Equation
data Equation    = Equation VarId [Pattern] Expr
data ConSig      = Sig ConId Type
data Pattern     = VarPat VarId
                 | ConPat ConId [Pattern]
data Kind        = Star | Arrow Kind Kind
data Type        = TypeVar VarId | TypeConst DataId
                 | TypeApp Type Type
                 | TypeFun Type Type
data Expr        = Var VarId | Con ConId
                 | App Expr Expr
```

Figure 2. Haskell syntax of the core language

list constructors of open data types, provide type signatures for normal functions, provide type signatures for open functions. Finally, functions are defined by listing one or more defining equations.

Such defining equations have a left hand side consisting of several patterns, but there are no guards. Patterns are reduced to variable and constructor patterns.

Kinds are only used in data type declarations.

The type and expression language are reduced as far as possible, only leaving identifiers and application (and function types).

Note that only top-level entities can be declared as open. Because of this restriction, local definitions (**let**) do not add anything interesting to the core language, hence we omit them.

There are several assumptions that a valid program must fulfil besides adhering to the correct syntax. There must be a distinguished module called *Main*, and the program must not contain modules that are not reachable from *Main* in the import graph.

All function cases for non-open functions must appear consecutively in a single module. Constructors that appear in a data type declaration must have the correct result type, and constructors appearing outside of data type declarations must have an open data type (that is in scope) as a result type. Constructor patterns must be fully applied.

All function cases for the same function (open or not) must have the same number of left hand side arguments. This restriction is somewhat arbitrary in Haskell (mainly in place to prevent errors), but it is essential to implement best-fit pattern matching for open functions, which only analyses left hand side arguments.

All identifiers used on the right hand sides of type signatures and function definitions must be in scope, and also all constructors that appear in patterns. A data declaration (and an open data declaration) brings the name of the data type into scope. A constructor signature brings the name of the constructor (for an open data type) into scope. A type signature of an open function brings the name of the open function into scope. A function definition for a non-open function brings the name of the function into scope. An **import** statement brings all entities (functions and data types) from the imported module into scope, and a module exports all entities it defines or imports.

We assume that all identifiers uniquely refer to a single entity in scope, and to emphasize this point, we assume that all identifiers are qualified with a module name. Defining occurrences of identifiers must point to the current module, i.e., $M.x$ can only be defined in module M. The only exception are equations of open functions. If new equations are added to an open function that is imported from another module, the new equations define the same entity, with the module identifier of the original module.

In Figure 3, we display the evaluator for expressions in the core language. The definitions of the open data type *Expr* and the open function *eval* are distributed over two modules *Expr* and *Main*. The left hand side shows the program in an informal style as it might be written by hand. The right hand side shows the same program with unique qualified identifiers (it is not necessary to qualify local variables; we use unqualified local variables for reasons of space).

In Section 4.4, we will discuss issues that arise if we consider the full Haskell language instead of our core language.

4.2 Semantics of open data types and open functions

We now define the semantics of open data types and open functions by mapping a program of the core language to another program in the same language that does not make use of open definitions (in other words, the resulting program does not contain occurrences of the data constructors *OpenDataDecl*, *Constructor*, or *OpenTypeSig*).

The transformation is straightforward. The resulting program contains only a single module *Main*, which comprises all entities of all modules from the original program. Names in the original program are always qualified. We therefore mangle an entity $M.x$ into an identifier local to module *Main* that contains both M and x. We write such a mangled identifier as *Main.M-x* in this paper. Each open data type is mapped to an ordinary data type. All constructors of the open data type are collected and appear as constructors of the closed data type in the target program. Similarly for functions, we group all cases that belong to an open function, effectively turning it into a closed function.

The result of translating the example program of Figure 3 is depicted in Figure 4.

Figure 5 formalizes the idea as a function *translateDecls*, which takes a single declaration to a list of declarations. As additional inputs, the function *translateDecls* expects a list of all constructors of open data types in the program, and a list of all equations of open functions in the program. Import statements and constructor type signatures are removed. They do not occur in the resulting program. A declaration of an open data type is translated into a declaration of an ordinary data type. We traverse the list of all constructors to collect the constructors for the data type in question. We proceed similarly for open functions: an open type signature is replaced by an ordinary type signature, and we collect all defining equations for that function. All other forms of declarations are unaffected by the translation.

The only tricky part of the process is how we order the collected constructors and function equations. Let us first consider constructors. The order of constructors in a data type definition has no effect in our example language. The order is, however, relevant in full Haskell: some of the type class instances that can be generated automatically by the compiler via the **deriving** construct (*Ord*, *Enum*, and *Bounded*) are affected by the textual order of constructors. Because we want to use the core language as a representative example for full Haskell, we specify the order *ccmp* ("constructor compare") in the following paragraph, and assume that the *allSigs* argument of *translateDecls* is sorted according to *ccmp*.

Consider the graph where each module that makes up the original program is a node, and each **import** statement defines a directed edge. The relative order between **import** statements in a module defines an order between the outgoing edges of a node. An order between modules is induced by a depth-first traversal of the graph, where the children are smaller than the parent, starting at *Main*

```
module Expr where
import Prelude

open data Expr :: *
Num :: Int → Expr

open eval :: Expr → Int
eval (Num n) = n

module Main where
import Prelude
import Expr

Plus :: Expr → Expr → Expr
eval (Plus e1 e2) = eval e1 + eval e2

module Prelude where
data Int :: * where {...}
(+) :: Int → Int → Int
(+) = ...
```

```
module Expr where
import Prelude

open data Expr.Expr :: *
Expr.Num :: Prelude.Int → Expr.Expr

open Expr.eval :: Expr.Expr → Prelude.Int
Expr.eval (Expr.Num n) = n

module Main where
import Prelude
import Expr

Main.Plus :: Expr.Expr → Expr.Expr → Expr.Expr
Expr.eval (Main.Plus e1 e2) = Expr.eval e1 Prelude.+ Expr.eval e2

module Prelude where
data Prelude.Int :: * where {...}
(Prelude.+) :: Prelude.Int → Prelude.Int → Prelude.Int
(Prelude.+) = ...
```

Figure 3. Expression problem in the core language

```
module Main where

data Main.Expr-Expr :: * where
  Main.Expr-Num :: Main.Prelude-Int → Main.Expr-Expr
  Main.Main-Plus :: Main.Expr-Expr → Main.Expr-Expr → Main.Expr-Expr

Main.Expr-eval :: Main.Expr-Expr → Main.Prelude-Int
Main.Expr-eval (Main.Expr-Num n)       = n
Main.Expr-eval (Main.Main-Plus e1 e2) = Main.Expr-eval e1 Main.Prelude-+ Main.Expr-eval e2

data Main.Prelude-Int :: * where {...}
(Main.Prelude-+) :: Main.Prelude-Int → Main.Prelude-Int → Main.Prelude-Int
(Main.Prelude-+) = ...
```

Figure 4. Example translation of expression evaluator

```
translateDecls :: [ConSig] → [Equation] → Declaration → [Declaration]
translateDecls allSigs allOpenEquations d =
  case d of
    ImportDecl _       → []
    OpenDataDecl n k → [DataDecl n k (filter (constrForData n) allSigs)]
    Constructor _      → []
    OpenTypeSig n t    → TypeSig n t :
                         [Function f | f ← allOpenEquations, nameOf f == n]
    Function f         → if nameOf f ∈ map nameOf allOpenEquations
                         then [] else [Function f]
    _                  → [d]
```

Figure 5. Semantics of declarations

$$
\begin{aligned}
&\mathit{pcmp} :: [\mathit{Pattern}] \to [\mathit{Pattern}] \to \mathit{Ordering}\\
&\mathit{pcmp}\ \mathit{ps}\ \mathit{ps}' = \mathit{foldr}\ (\lhd)\ \mathit{EQ}\ (\mathit{zipWith}\ (\sim)\ \mathit{ps}\ \mathit{ps}')\\
&(\sim) :: \mathit{Pattern} \to \mathit{Pattern} \to \mathit{Ordering}\\
&\mathit{VarPat}\ \_ \quad \sim \mathit{ConPat}\ \_\ \_ \quad = \mathit{LT}\\
&\mathit{ConPat}\ \_\ \_ \sim \mathit{VarPat}\ \_ \quad = \mathit{GT}\\
&\mathit{VarPat}\ \_ \quad \sim \mathit{VarPat}\ \_ \quad = \mathit{EQ}\\
&\mathit{ConPat}\ n\ \mathit{ps} \sim \mathit{ConPat}\ n'\ \mathit{ps}' = \mathit{ccmp}\ n\ n' \lhd \mathit{pcmp}\ \mathit{ps}\ \mathit{ps}'\\
&(\lhd) :: \mathit{Ordering} \to \mathit{Ordering} \to \mathit{Ordering}\\
&\mathit{LT} \lhd \_ = \mathit{LT}\\
&\mathit{EQ} \lhd x = x\\
&\mathit{GT} \lhd \_ = \mathit{GT}
\end{aligned}
$$

Figure 7. Best-fit ordering

(i.e., *Main* is the largest element). The order *ccmp* is determined by the location of the data constructors to compare: if both are in the same module, we use the relative order of statements; if the data constructors reside in different modules, we use the module order. Note that even though *ccmp* can be slightly difficult to track for the programmer, the important fact is that there exists a well-specified order, so that the above-mentioned classes can be derived.

For functions, however, the situation is different. We will explain in Section 4.3 how we order the equations of open functions.

4.3 Best-fit pattern matching

In Section 2.5, we have argued that *best-fit* pattern matching is preferable to *first-fit* pattern matching in the context of open functions. Let us now formalize the semantics of best-fit pattern matching. As outlined in Section 2, our proposed approach is inspired by technology that is available for type classes, namely the resolution of overlapping instances.

To summarize, the idea is that we take not the first matching case, but the best matching case. A value of constructor *C* matches a pattern of constructor *C better* than a pattern that is a variable. If *C* has arguments, then the arguments are matched according to the same policy from left to right. It is an error if the exact same pattern occurs twice for the same open function.

Figure 6 shows an example. The left hand side lists the equations of a function f in an order in which they might appear in the program. The right hand side shows a closed version of f where the equations of f have been reordered so that the best-fit semantics and Haskell's first-fit semantics coincide.

Such a reordering of the equations can be computed by defining an ordering on lists of patterns as given in Figure 7. Lists of patterns are ordered lexicographically using $(\lhd)$, where the order on individual patterns is given by $(\sim)$ as follows: variable patterns are smaller than constructor patterns. If we have to compare two constructor patterns, we compare the constructor names and the argument patterns and combine the results lexicographically. The order on constructor names is irrelevant (because patterns of different constructors never overlap), so we choose the constructor ordering *ccmp*.

If the function *translateDecls* from Figure 5 receives the list *allOpenEquations* sorted according to the specified order, then the resulting program will contain the equations of open functions in such an order that best-fit and first-fit pattern matching coincide, as in the right hand side of Figure 6.

4.4 Haskell

Let us now delve into the finer points of our proposal, and consider all of Haskell rather than our core language.

Module system The Haskell module system allows to specify export and import lists, and it allows to import modules qualified and under a different name. We have the following design choice for open data types and functions:

- open functions, open data types, and constructors of open data types are always exported and cannot be hidden, or
- open entities can selectively be hidden, but that only controls their visibility, without affecting the semantics. If a constructor of an open data type is not visible in a certain module, it cannot be referenced by name, but it still exists.

Qualified imports and module renamings only complicate the rules of how identifiers refer to entities, but they do not interact with open data types and functions.

Local bindings As we have explained before, all open entities are top-level entities. Local (let-bound) function definitions cannot be open, even if they have a local name. We do not think that this is a problem, because functions that should be open can be lifted to the top level.

Type classes Although we used type classes for inspiration, there is almost no interaction between type classes and open entities. The **deriving** construct is one exception. Deriving type class instances for open data types is possible. The **deriving** clause can be specified at the point where the open data type is introduced. The semantics is that the translated normal data type contains the same **deriving** clause. A useful variation for open data types would be that derived type classes can be specified separately from the **open data** definition. A statement such as

derive *Show Expr*

could ask the compiler to equip the open data type *Expr* with a *Show* instance at a later point.

Type class instances for open data types can be defined normally. If a class method is desired to be open, it must be defined to be equal to an open top-level function.

Patterns and guards Haskell's pattern language is more expressive than the patterns of the core language. Let us consider each of the different pattern constructs in turn:

As-patterns do not affect the semantics of open functions at all. An as-pattern is ordered like that pattern with the as-clauses removed.

Wildcard patterns (_) are syntactic sugar for variable patterns and are thus treated like variable patterns.

Irrefutable patterns are matched lazily. In other words, the match always succeeds at first; the value is only actually decomposed if the components are accessed later (leading to potential run-time failure at that point). For the purpose of selecting the right equation of a function, an irrefutable pattern behaves as a variable pattern (the match always succeeds), and we therefore treat it like a variable.

Guards in Haskell interact with pattern matching. Guards are boolean conditions that can be attached to the left hand side of a function. An equation only matches if the patterns of the equation match and the guard succeeds. Otherwise, the next function equation is matched.

If guards are added to the pattern language, it is somewhat unclear what "the next function equation" means in the context of best-fit pattern matching. There are several options to deal with this problem:

- Completely ignore the guards to determine the best-fitting equation (and possibly raise a run-time error if all guards of an otherwise matching equation fail).

```
open f :: [Int] → Either Int Char → ...
f (x:xs) (Left 1)      = ...
f y      (Right a)     = ...
f (0:xs) (Right 'X')   = ...
f (1:[]) z             = ...
f (0:[]) z             = ...
f []     z             = ...
f (0:[]) (Left b)      = ...
f (0:[]) (Left 2)      = ...
f y      z             = ...
f (x:[]) z             = ...
```

```
f :: [Int] → Either Int Char → ...
f []     z             = ...
f (0:[]) (Left 2)      = ...
f (0:[]) (Left b)      = ...
f (0:[]) z             = ...
f (0:xs) (Right 'X')   = ...
f (1:[]) z             = ...
f (x:[]) z             = ...
f (x:xs) (Left 1)      = ...
f y      (Right a)     = ...
f y      z             = ...
```

Figure 6. Example of the semantics of best-fit left-to-right pattern matching

- Collect guards in equations for the same patterns according to a variant of the *ccmp* order (the order in which they appear in the program, inlining **import** statements).
- Disallow guards in equations for open functions.

5. Implementation

The goal of this section is to sketch two implementations of open data types and functions. One is directly based on the semantics, the other allows separate compilation, but requires mutually recursive modules.

5.1 Naïve implementation

The semantics of open data types and open functions defined in Section 4 is defined as a source-to-source transformation where the resulting program does not contain any open data types or open functions. The Figures 3 and 4 illustrate the transformation for the evaluator on expressions. This translation constitutes already a possible implementation, and it has the advantage of being correct by construction.

The disadvantage of this approach is that a whole program must be compiled at once, i.e., there is no separate compilation of modules. If any single module changes, the transformation has to be applied again and the entire program must be compiled again. This removes a bit of the just gained flexibility, because compilation times of resulting programs can be high, and libraries containing open data types or functions must be distributed in source form.

As a pragmatic optimization, modules that do not define or use open data types or functions (for example, the Haskell *Prelude*) can be excluded from the collapsing process.

It is important to realize that while compilation times may be higher when the whole program is compiled at once, the resulting programs are probably more efficient. Experience with the tool Haskell All-In-One [10], which performs the process of collapsing a Haskell program into one module (without any further support for open data types and functions, however), suggests that resulting programs are usually faster than programs compiled via separate compilation, because compilers can apply more optimizations if the complete program information is available at every point.

The new Haskell compiler Jhc [11] is also based on this assumption and compiles whole programs at once for better optimization. Jhc can cache module information in an internal format, and compilation times are acceptable for moderately-sized applications. We expect that open data types and open functions would be easy to add to Jhc using the naïve implementation technique.

5.2 Implementation that supports separate compilation

We now present an alternative translation scheme for open data types and open functions that supports separate compilation. A look back at the naïve implementation immediately reveals a simple opportunity for optimization: only because we have open data types and open functions in some parts of the program, we have entirely collapsed the program into a single module. But the idea to collect open data types and open functions and translate them into closed data types and closed functions only affects the declarations of such open entities, which probably constitute a relatively small part of the entire program. A better approach is thus to translate each module M to a module M' which contains all the original declarations except the declarations of open entities. In particular, *Main* is translated to *Main*$'$ and if M imports N, then M' imports N'. The resulting program consists of an additional module, a new *Main* module, which contains all the collected open entities of the program.

The modules of the resulting program are usually mutually recursive: each module M that defines an open entity results in a module M' that imports *Main*; but *Main* in turn depends on many modules of the program, because it contains the definitions of all the open entities.

The above solution has two disadvantages:

- The code of open functions is not compiled separately. Even if the structure of an open function is unchanged and only a single right hand side of one equation of that function is modified, the entire module *Main* has to be recompiled.
- While a Haskell implementation supporting mutually recursive modules can now compile the modules separately, it cannot compile them independently. Whenever a definition of an open entity changes, all modules depending on *Main* must be recompiled.

In the following, we explain how these two problems can be alleviated. Figure 8 shows the result of translating the program from Figure 3 according to the final version of this translation scheme.

Splitting left and right hand sides of functions An equation of a function can be split into two equations: the first performs the pattern matching and in the case of success calls the other; the second executes the right hand side.

As an example, consider the equation for *eval* on the *Plus* constructor:

$$eval\ (Plus\ e_1\ e_2) = eval\ e_1 + eval\ e_2$$

This equation is split into the two following equations:

$$\begin{aligned} eval\ (Plus\ e_1\ e_2) &= eval'\ e_1\ e_2 \\ eval'\ e_1\ e_2 &= eval\ e_1 + eval\ e_2 \end{aligned}$$

The second equation no longer performs pattern matching. It has only variables as arguments: the free variables that occur in the patterns of the original function. The second equation thus stands alone and can be compiled separately, whereas the first must constitute a part of the final *eval* function that contains all patterns.

```
module Expr' where
import Prelude
import Main
  (Main.Expr-Expr  :: *,
   Main.Expr-Num   :: Prelude.Int → Expr,
   Main.Expr-eval  :: Main.Expr-Expr → Prelude.Int)
Expr.Expr-eval_1 n = n

module Main' where
import Prelude
import Expr'
import Main
  (Main.Expr-Expr  :: *,
   Main.Expr-Num   :: Prelude.Int → Expr,
   Main.Main-Plus  :: Main.Expr-Expr → Main.Expr-Expr → Main.Expr-Expr,
   Main.Expr-eval  :: Main.Expr-Expr → Prelude.Int)
Main.Expr-eval_2 e_1 e_2 = Main.Expr-eval e_1 Prelude.+ Main.Expr-eval e_2

module Main where
import Expr'
import Main'
data Main.Expr-Expr :: * where
  Main.Expr-Num :: Main.Prelude-Int → Main.Expr-Expr
  Main.Main-Plus :: Main.Expr-Expr → Main.Expr-Expr → Main.Expr-Expr
Main.Expr-eval :: Main.Expr-Expr → Prelude.Int
Main.Expr-eval (Main.Expr-Num n)       = Expr.eval_1 n
Main.Expr-eval (Main.Main-Plus e_1 e_2) = Expr.eval_2 e_1 e_2
```

Figure 8. Example translation of expression evaluator supporting separate compilation

The general technique is the following: If module M contains an open function equation of the form

$$N.f\ p_1 \ldots p_n = rhs$$

and if $v_1, \ldots, v_k$ are the variables bound by the patterns $p_1, \ldots, p_n$, we generate two equations for the target program:

$$\begin{aligned} &Main.N\text{-}f\ p_1 \ldots p_n = M'.M\text{-}f_u\ v_1 \ldots v_k \\ &M'.M\text{-}f_u\ v_1 \ldots v_k \quad = rhs \end{aligned}$$

Here, u is a module-wide unique number, to distinguish different equations of one function $M'.f$ defined in a single module. The second equation is placed as a function definition in the translated module M', whereas the first equation constitutes part of a function definition in module *Main*.

Stable module interfaces While many modules import *Main*, most modules use only a small part of it. The functionality that a module M' expects *Main* to provide can be captured in an *interface*. The interface changes only when the original module M is modified, because the interface comprises just the open data types, constructors, and open functions that are in scope in M.

On the other hand, as long as the interface between *Main* and M remains stable, it is reasonable to assume that an implementation does not have to recompile M when *Main* changes.

In the example in Figure 8, we have attached the interface to the import statements. The syntax of the core language must be extended as shown in Figure 9.

An interface consists of many interface declarations. Each interface declaration can be the kind signature of a data type, a type signature of a constructor, or a type signature of a function. The imported module *implements* the interface if it exports the specified entities with the specified kinds or types. A program is valid only if all specified interfaces are implemented by the imported modules.

GHC implements almost all the features we describe here. It supports mutually recursive modules, and it supports the declaration of stable interfaces in such a way that a module only has to be recompiled if the interface changes. Interfaces have to be specified by the programmer in GHC-specific `.hs-boot` files, which are written in a subset of Haskell.

What GHC does not currently support is to specify constructors as single entities within an interface. Data types can only be entirely abstract (not allowing pattern matching) or concrete with all constructors. The reason is that pattern matching can be compiled more efficiently if the layout of the data type is known completely. There are no theoretical difficulties in lifting this restriction, but it might imply a small performance loss if closed functions pattern match on open data types.

6. Related work

6.1 The expression problem

The expression problem which was originally posted by Wadler [1] has received a lot of attention, and an impressive amount of research has been performed in order to solve the problem.

declaration $decl ::= \textbf{import}\ moduleid\ (idecl^*) \mid \ldots$
interface declaration $idecl ::= \textbf{data}\ dataid :: kind \mid consig \mid varid :: type$

Figure 9. Syntax of the core language for open data types and open functions

Zenger and Odersky [12] present a list of criteria to evaluate solutions to the expression problem:

- extensibility for both data types and functions,
- strong static type safety,
- no modification or duplication of existing code,
- and separate compilation.

Our solution – using the translation based on mutually recursive modules – fulfills all four properties.

Strong static type safety can be implemented to various degrees: Haskell's type system automatically ensures that functions are only applied to values of the correct (open) data type, but it does not guarantee the absence of pattern match failures, which usually are runtime errors in Haskell. Haskell implementations such as GHC check exhaustiveness of pattern matching and issue a compile-time warning, but transferred to our approach, this means that the exhaustiveness check is global for the open functions only occurs when *Main* is compiled, and is therefore not modular.

Millstein *et al.* [3] have a solution to the expression problem for ML that is in some ways very close to ours: their approach is also based on both open data types (hierarchical classes in their language) and open functions, and their syntax can be mapped to ours almost one-to-one. In their system, there is no distinction between open and closed entities, making it possible to extend anything at any time without a major impact on the program. Function cases are also ordered according to a form of best-fit pattern matching. However, their approach is both stronger and more complex than ours: they support "implementation-side type checking", i.e., their system can statically and modularly check that all pattern matches are exhaustive and unambiguous; on the other hand, their system adds classes and inheritance (and thus a form of subtyping to the language), and places a couple of restrictions on the patterns of open functions.

Most of the work on the expression problem is focused on OO languages. A noteworthy example is the one of Zenger and Odersky [13], because their solution is very similar to ours, albeit in an OO language: algebraic data types and pattern matching are added to an OO language, thereby providing a mixture of object-oriented and functional constructs within a single language. By integrating subclassing with algebraic data types, such data types can be extended by new constructors. Open functions can then also be modelled by subclassing and redefining a method.

Both Zenger and Odersky [12, 13] as well as Torgersen [14] compare and categorize many proposed solutions. Most approaches to the expression problem are based on rather heavyweight type system extensions, such as mixins [4] or multi-methods [2, 3] and subtyping, which is a central feature in (usually type-checked) OO feature, but not trivial to add to functional languages (see also below).

6.2 Polymorphic variants, extensible variants

Polymorphic variants are implemented in OCaml [15] and have been proposed for Haskell [16, 17]. *Variant types* are anonymous types that enumerate a number of constructors. If one variant's constructors are a subset of another, the two variant types are in a subtype relationship. The programmer can thus extend a variant with a new constructor, and existing functions continue to work. Garrigue [18] shows that variants can solve the expression problem. Variants do not, however, yield open functions: if a function is supposed to work on an additional constructor, the programmer must adapt the original definition, or define a new function that adds the new case and otherwise refers to the old version.

If open functions are simulated by the definition of a new wrapper function, recursive calls in the original function will not point to the new wrapper, but continue to point to the original function. Recursion must therefore be tied manually using a fixed-point operator at appropriate places, causing more work for the programmer.

Variants are by no means a simple language extension, and complicate the type system significantly. Gaster and Jones [16] show that extensible variants are related to extensible records.

6.3 O'Haskell, OOHaskell

Several attempts have been made to add object-oriented programming capabilities to Haskell. O'Haskell [19] is a language extension that adds subtyping to Haskell's type system and introduces objects as a special language construct. OOHaskell [20] is a library and a collection of coding techniques that enable to simulate object-oriented programming in current Haskell (with some widely used extensions), using the type class system.

The main difference of this line of work in contrast to ours is that these approaches are far more ambitious: they aim at allowing object-oriented techniques in Haskell, whereas we only want to add a new direction of extensibility.

While the two approaches may allow solutions to the expression problem involving objects, they have the significant drawback that they force an object-oriented mind-set on the programmer in order to do so: if we discover that a closed data type should in fact be open, the program has to be restructured such that the data type is mapped to a class. In contrast, declaring the data type as open can be achieved by adding a keyword **open** and removing a keyword **where** in order to 'free' the constructors – a very local change.

6.4 Type classes

Even without the OOHaskell machinery, it is simple to simulate open data types and functions using type classes. While the same argument just made – restructuring of the program and syntactical overhead – holds for this encoding, too, it is nevertheless worthwhile to explore the connection a bit further, because we have used ideas from type classes throughout this paper to guide us in our design for open data types and open functions.

When using a type class encoding, open data types are mapped to several data types, one per constructor, and open functions are mapped to type classes with a single method. Figure 10 shows the evaluator on expressions rewritten in this style as an example.

This encoding bears several disadvantages:

- If we use type classes, we cannot define open functions in a natural syntax with the full powers of pattern matching. This disadvantage is most obvious if we reconsider the exception example from Section 3.2. An exception handler is usually a local function pattern matching on a limited number of exception constructors. It would be extremely inconvenient to lift each handler to a top-level class definition, with each branch of the **case** statement being an instance of that class.
- Data type definitions become awkward, because constructors have to be lifted to types. There is no clear relation anymore between a constructor and the data type it constructs. If constructors are data types, we cannot express in Haskell that a function

```
data Num      = Num Int
class Eval a
  where eval            :: a → Int
instance Eval Num
  where eval (Num n)    = n
data Plus a b = Plus a b
instance (Eval a, Eval b) ⇒ Eval (Plus a b)
  where eval (Plus a b) = eval a + eval b
```

Figure 10. Extensible evaluator on expressions using type classes

such as *Eval* expects only constructors of a certain data type. This weakness could be eliminated by introducing *data kinds* as in Omega [21].

- By turning constructors into data types, the type system is likely to get in the way: the compiler wants to keep track of which constructor a value belongs to at all times, and if that choice is dependent on the outcome of a run-time computation, this is impossible.

There are, however, also advantages of the type class encoding that we have not yet exploited in our design of open data types. Because constructors live on the type level, there is improved type safety: for each call site of a class method, the compiler checks if an appropriate instance is available. In the example, if we call the *eval* method on a *Plus* value, the compiler will actually verify that there is an instance of class *Eval* for type *Plus*, and if such an instance does not exist, a compile-time error is triggered. Calls to open functions are only checked for type correctness. But for open functions on open data types, pattern match failures are not unlikely, and only revealed at runtime.

A possibility to get a static check without lifting constructors to the type level is to perform the check whenever the constructor of a function argument is known at compile time, and to issue warnings if a pattern match failure cannot be ruled out statically. There is ongoing work by Mitchell and Runciman to develop a system that performs such checks for the Haskell language [22]. Such a system can be used independently of open data types and open functions, but we expect it to be particularly useful in this context.

Another feature of type classes is that the correct dictionary argument is inferred automatically. As it turns out, we can use the same technique to infer the correct argument to both closed and open functions in some situations. Recall the application of open functions to generic programming from Section 3.1. An overloaded function is implemented as an open function with an argument of type *Type*. This type of type representations is a GADT with the property that for any choice of a, *Type* a is a *singleton type*, i.e., a type with only one value (modulo the use of $\perp$). We can infer this type representation automatically in many cases. It is possible to program the class plus instances out by hand, but given the definition of the *Type* GADT, the class and instance declarations follow the structure of the GADT precisely, and could be generated automatically by a compiler. If generalized in such a way that arguments can be inferred if there is exactly one way to construct them, this technique constitutes a way to implement *explicit implicit parameters* [23]. Dependently typed systems like Epigram [24] can sometimes infer pieces of code if they are uniquely determined by the context.

6.5 Pattern matching

Haskell's first-fit pattern matching semantics are not suitable for open functions, because first-fit relies on the relative order of the defining equations, which is hard to track for open functions.

We have therefore proposed best-fit pattern matching as an alternative pattern matching semantics. Our use of best-fit pattern matching in the context of open data types is inspired by the resolution mechanism for overlapping instances and by the support of nondeterministic functions in the functional-logical programming language Curry [25], where all equations that match are chosen. Best-fit pattern matching (without the left-to-right bias, and without connections to open data types) is described by Field and Harrison [26]. Best-fit pattern matching is also related to method dispatch in OO languages, and the relation becomes more pronounced in the context of multi-methods [2, 3].

First-class patterns [27] allow the programmer to choose different forms of pattern matching for different functions. This comes at the price of complexity, and it seems as if first-fit and best-fit pattern matching are sufficient for many applications.

7. Conclusions

We have presented a lightweight solution to the expression problem by proposing the addition of open data types and open functions to the Haskell language. The semantics of open data types and open functions is stunningly simple, and we firmly believe that this is a good thing: in the related work, many of the approaches either aim at far higher goals or consist of much more radical changes to the underlying language.

In contrast, our proposal does not have any consequences for the type system of Haskell, and it does not affect the semantics of other Haskell constructs. Furthermore, open data types provide a solution to the expression problem that fits well into the functional programming paradigm: we can continue to work with algebraic data types and to write functions that are defined via pattern matching.

A key concept is the introduction of best-fit pattern matching, which provides a way to compose several defining equations of a function without resorting to order of appearance in the code. We think that best-fit pattern matching is easy to grasp and adequate for open functions, as a similar mechanism is used already to resolve overlapping type class instances.

Haskell's support for mutually recursive modules turned out to be essential to achieve separate compilation for open functions. It would be interesting to compare this situation with separate compilation approaches for the expression problem in OO languages, but we have not done so yet.

We have presented several applications of open data types and open functions. We expect that open data types are also useful in the context of dynamic applications [28].

Acknowledgments

We thank Bruno Oliveira for the enlightening and fruitful discussions on the topic of open data types during his visit to Bonn, and John Reppy and Dave Herman for pointing out related work. We also thank the anonymous referees for their detailed and constructive comments. Special thanks go to Manuel Chakravarty for countless insightful remarks and suggestions. This research was supported by the DFG grant "A generic functional programming language".

References

[1] Wadler, P.: The expression problem. Mail to the java-genericity mailing list (1998)

[2] Chambers, C.: Object-oriented multi-methods in Cecil. In: ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag (1992) 33–56

[3] Millstein, T., Bleckner, C., Chambers, C.: Modular typechecking for hierarchically extensible datatypes and functions. In: ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, ACM Press (2002) 110–122

[4] Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1998) 171–183

[5] Hinze, R.: Generics for the masses. In Fisher, K., ed.: Proceedings of the 2004 International Conference on Functional Programming, Snowbird, Utah, September 19–22, 2004. (2004) 236–243

[6] Löh, A., Jeuring, J., Clarke, D., Hinze, R., Rodriguez, A., de Wit, J.: The Generic Haskell user's guide, version 1.42 (Coral). Technical Report UU-CS-2005-004, Institute of Information and Computing Sciences, Utrecht University (2005)

[7] Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, ACM Press (2002) 90–104

[8] Oliveira, B.C., Gibbons, J.: TypeCase: A design pattern for type-indexed functions. In: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, ACM Press (2005) 98–109

[9] Hinze, R., Löh, A., Oliveira, B.C.: "Scrap Your Boilerplate" reloaded. In Hagiya, M., Wadler, P., eds.: Proceedings of FLOPS 2006. Lecture Notes in Computer Science, Springer (2006) Available from `http://www.iai.uni-bonn.de/~loeh/SYB0.html`.

[10] Daume, H.: Haskell all-in-one. Tool homepage (2003) `http://www.isi.edu/~hdaume/HAllInOne/`.

[11] Meacham, J.: Jhc. Compiler homepage (2006) `http://repetae.net/john/computer/jhc/jhc.html`.

[12] Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: Foundations of Object-Oriented Languages (FOOL 2005). (2005)

[13] Zenger, M., Odersky, M.: Extensible algebraic datatypes with defaults. In: Proceedings of the International Conference on Functional Programming (ICFP 2001). (2001)

[14] Torgersen, M.: The expression problem revisited – four new solutions using generics. In Odersky, M., ed.: Proceedings of ECOOP 2004, Springer (2004) 123–146

[15] Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system release 3.09 – Documentation and user's manual. Institut National de Recherche en Informatique et en Automatique. (2004) Available from `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[16] Gaster, B.R., Jones, M.P.: A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham (1996)

[17] Leijen, D.: Extensible records with scoped labels. In: Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05). (2005) 297–312

[18] Garrigue, J.: Code reuse through polymorphic variants. In: Workshop on Foundations of Software Engineering. (2000)

[19] Nordlander, J.: A survey of O'Haskell. Web article (2001) Available from `http://www.cs.chalmers.se/~nordland/ohaskell/survey.html`.

[20] Kiselyov, O., Lämmel, R.: Haskell's overlooked object system. Draft. Available from `http://homepages.cwi.nl/~ralf/OOHaskell/` (2005)

[21] Sheard, T.: Putting Curry-Howard to work. In: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, ACM Press (2005) 74–85

[22] Mitchell, N., Runciman, C.: Unfailing Haskell: a static checker for pattern matching. In: Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05). (2005) 313–328

[23] Dijkstra, A., Swierstra, S.D.: Explicit implicit parameters. Technical Report UU-CS-2004-059, Institute of Information and Computing Sciences, Utrecht University (2004)

[24] McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming **14**(1) (2004) 69–111

[25] Hanus, M.: Curry – An Integrated Functional Logic Language (Version 0.8). (2003) Available from `http://www.informatik.uni-kiel.de/~mh/curry/report.html`.

[26] Field, A.J., Harrison, P.G.: Functional Programming. Addison-Wesley (1988)

[27] Tullsen, M.: First class patterns. In Pontelli, E., Costa, V.S., eds.: Practical Aspects of Declarative Languages, Second International Workshop. Number 1753 in Lecture Notes in Computer Science, Springer (2000) 1–15

[28] Stewart, D., Chakravarty, M.M.T.: Dynamic applications from the ground up. In: Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, ACM Press (2005) 27–38