Domain Specific Embedded Compilers

Daan Leijen and Erik Meijer
University of Utrecht
Department of Computer Science
POBox 80.089, 3508 TB Utrecht, The Netherlands
{daan, erik}@cs.uu.nl, http://www.cs.uu.nl/~{daan, erik}

Abstract

Domain-specific embedded languages (DSELs) expressed in higher-order, typed (HOT) languages provide a composable framework for domain-specific abstractions. Such a framework is of greater utility than a collection of stand-alone domain-specific languages. Usually, embedded domain specific languages are build on top of a set of domain specific primitive functions that are ultimately implemented using some form of foreign function call. We sketch a general design pattern for embedding client-server style services into Haskell using a domain specific embedded compiler for the server's source language. In particular we apply this idea to implement Haskell/DB, a domain specific embdedded compiler that dynamically generates of SQL queries from monad comprehensions, which are then executed on an arbitrary ODBC database server.

1 Introduction

Databases are ubiquitous in computer science. For instance, a web site is usually a fancy facade in front of a conventional database, which makes the information available in a convenient browsable form. Sometimes, servers are even running directly on a database engine that generates pages from database records on-the-fly. Hence is is not surprising that database vendors provide hooks that enable client applications to access and manipulate their database servers. On Unix platforms this is usually done via ODBC, under Windows there are confusingly many possibilities such as ADO, OLE DB, and ODBC.

What is common to all the above database bindings is that queries are communicated to the database as unstructured strings (usually) representing SQL expressions. This low-level approach has many disadvantages:

- Programmers get no (static) safeguards against creating syntactically incorrect or ill-typed queries, which can lead to hard to find runtime errors.
- Programmers have to distinguish between at least two different languages, SQL and the scripting language that generates the queries and submits them to the database engine (Perl, Visual Basic). This makes programming needlessly complex.
- Programmers are exposed to the accidental complexity and idiosyncrasies of the particular database binding, making code harder to write and less robust against the vendor's fads [1].

We argue that domain-specific embedded languages [9] (DSELs) expressed in higher-order, typed (HOT) languages, Haskell [10] in our case, provide a composable framework for domain-specific abstractions that is of greater utility than a collection of stand-alone domain-specific languages:

- Programmers have to learn only one language, domain specific abstractions are exposed to the host language as extension libraries.
- In many cases it is possible to present libraries using a convenient domain specific syntax.
- It is nearly always[1] possible to guarantee that programmers can only produce syntactically correct target programs, and in many cases we are able to impose domain specific typing rules.

[1] For instance $\perp$ is a value of every type in Haskell, so cannot prevent programmers from producing infinite or partially defined values.

- Programmers can seamlessly integrate with other domain specific libraries (e.g. CGI, mail), which are accessible in the same way as any other library. This is a largely underestimated benefit of using the embedded approach. Connecting different domain specific languages together is usually quite difficult.
- Programmers can leverage on existing language infrastructure such as the module and type system and the built-in abstraction mechanisms.

Note that the ideas underlying our thesis date way back to 1966 when Peter Landin [12] already observed that all programming languages comprise a domain independent linguistic framework and a domain dependent set of components. What is new in this paper is that we show how to embed the terms and the type system of another (domain specific) *programming language* into the Haskell framework, which dynamically *compiles* and *executes* programs written in the embedded language. Moreover, no changes to the syntax or additions of primitives were needed to embed the language in Haskell.

1.1 Overview

We begin by giving a minuscule introduction to Haskell and a crash course in relational databases, and we show how a typical Visual Basic and a typical Haskell program would access a relational database. Next we sketch a general design pattern for term- and type-safe embedding client-server style services into Haskell using an evaluator for a subset of SQL expressions as an example server. We then turn our attention to the more challenging task of embedding a database server in Haskell. Section 7 contrasts the Haskell and Visual Basic implementations of a example web-page that generates HTML from a database of exam marks. We finish with some conclusions and some ideas for future work.

2 Minuscule introduction to Haskell

The main virtue of a functional language is that functions are first-class citizens that can be stored in lists, or passed as arguments to and returned from other functions. To emphasize the fact that functions of type `a -> b` have the same status as any other kind of value, we usually write them as lambda-expressions `f = \a -> b` instead of the more common Haskellish notation `f a = b`.

Function application is given by juxtaposition in Haskell and associates to the left. Thus when we have the three argument function `line`:

```
line = \a -> \b -> \x -> a*x + b
```

the application `line 2 3` denotes the single argument function `\x -> 2*x + 3`. The type of the `line` function can be specified explicitly:

```
line :: Int -> Int -> Int -> Int
```

The type shows that `line` takes three `Int` arguments and returns an `Int`.

Case expressions are used to define functions by case distinction, for instance the factorial function can be defined in Haskell as

```
fac :: Num a => a -> a
fac = \n ->
  case n of
    { 0 -> 1
    ; n -> n * fac (n-1)
    }
```

In Haskell, polymorphic types can be constrained by means of type contexts. The given type for the factorial function `fac :: Num a => a -> a -> a` says that function `fac` has type `a -> a -> a` for all types `a` that are instances of the `Num` class. Unsurprisingly it is the case that `Num Int`, `Num Float` and `Num Double` are all true.

We will represent database rows by extensible records, an experimental feature that is currently only supported by the the *TREX* extension of the Hugs implementation of Haskell [8]. A record is nothing more than an association list of field-value pairs. For instance the record `(x = 3, even = False) :: Rec (x :: Int, even :: Bool)` has two fields, `x` of type `Int` and `even` of type `Bool`. A record of type `Rec r` can be *extended* by a field `z` provided that `z` does not already occur in `r`. The fact that record `r` should lack field `z` is indicated by the constraint `r\z`, thus the type of a function that adds an `foo` field to a record becomes:

```
extendWithFoo :: r\foo =>
```

```
  a -> Rec r -> Rec (foo :: a | r)
extendWithFoo = \a -> \r -> (foo = a | r)
```

Unfortunately, labels are not (yet) first class values in TREX, so we *cannot* write a generic function that extends a given record with a new field:

```
-- WRONG
extendWith = \(f,a) -> \r -> (f = a | r)
```

In our case the lack of first class labels means that we have to repeat a lot of code that only differs in the names of some labels. Another deficiency of the current implementation of TREX records is the fact that it is impossible to formulate a constraint on all the values in a given record, for instance, we would like to constrain a record to contain only values on which equality is defined. Currently, there is just one built-in constraint `ShowRecRow r` that indicates that all values in row `r` are in the `Show` class.

When interacting with the outside world or accessing object models, we have to deal with side-effects. In Haskell, effectful computations live in the IO monad [15]. A value of type `IO a` is a *latently* effectful computation that, *when executed*, will produce a value of type `a`. For instance, the command `getChar :: IO Char` will read a character from the standard input when it is executed.

Like functions, latently effectful computations are first class citizens that can be passed as arguments, stored in list, and returned as results. For example `putChar :: Char -> IO ()` is a function that takes a character and then returns a computation that, when executed, will print that character on the standard output.

Effectful computations can be composed using the `do{}`-notation. The command `do{ a <- ma; f a } :: IO b` is a latent computation, that, when executed, first executes the command `ma :: IO a` to obtain a value `a :: a`, passes that to the action-producing function `f :: a -> IO b` and then executes `(f a)` to obtain a value of type `IO b`. For example, when executed, the command:

```
do{ c <- getChar
  ; putChar c
  }
```

reads a character from the standard input and copies it to the standard output.

The usefulness of monads goes far beyond input/output, many other type constructors are monads as well. In section 6 we will define the `Query` monad that allows us to write queries using the same `do{}` notation that we introduced here for IO-computations.

In this paper we adopt style conventions that emphasize when we are dealing with effectful computations. Specifically, all expressions of monadic type (such as `IO` and `Query`) are written with an explicit `do{}`. To reflect the influence of the OO style, we will use postfix function application `object#method = method object` to mimic the `object.method` notation. Together with the convention for writing functions as lambda-expressions, this results in highly stylized programs from which it is easy to tell the type of an expression by looking at its syntactic shape.

3 A crash course in relational databases

In a relational database [5], data is represented as sets of tuples. For example take the following database *Rogerson* of objects and some of their properties [17]:

Object	Edible	Inheritance	President
Rich people	False	False	True
Bean plants	True	False	False
CORBA	False	True	False
COM	False	False	False

We can conclude from this table that bean plants are edible, and that rich people can run for president. We can query the database more systematically using relational algebra.

The *selection* operator σ specifies a subset of rows whose attributes satisfy some property. For example we can eliminate all entries for objects that can run for president from the database using the expression $\sigma_{President=False}\ Rogerson$:

Object	Edible	Inheritance	President
Bean plants	True	False	False
CORBA	False	True	False
COM	False	False	False

The *projection* operator π specifies a subset of the columns of the database. For example, we can return all objects that are edible using the query $\pi_{Object}\ (\sigma_{Edible=True}\ Rogerson)$

Object
Bean plants

Another typical operation on relations is join $\bowtie$ that combines two relations by merging tuples whose common attributes have identical values. Hence, if we join the *Presidents* table

Name	President
Starr	False
Clinton	True

with the *Rogerson* table using $\pi_{Name,Object}(Presidents \bowtie Rogerson)$ we get a table with the name and object description of people that can run for president:

Name	Object
Clinton	Rich people

3.1 The SQL way: `SELECT` statement

SQL is the defacto standard programming language to formulate queries over relational databases. The SQL query

```
SELECT columns
FROM tables
WHERE criteria
```

combines selections, projections and joins in one powerful primitive. The `SELECT` clause specifies the columns to project, the `FROM` clause specifies the tables where the columns are located and the `WHERE` clause specifies which rows in the tables should be selected.

The query $\sigma_{President=False}\ Rogerson$ is expressed in SQL as:

```
SELECT *
FROM Rogerson AS r
WHERE r.President = FALSE
```

The query $\pi_{Object}\ (\sigma_{Edible=True}\ Rogerson)$ becomes:

```
SELECT r.Object
FROM Rogerson AS r
WHERE r.Edible = TRUE
```

The query $\pi_{Name,Object}(Presidents \bowtie Rogerson)$ is expressed as:

```
SELECT p.name, r.Object
FROM Rogerson AS r, Presidents AS p
WHERE r.President = p.President
```

3.2 The VB way: Unstructured strings

The common way to do query processing from Visual Basic is to build an unstructured string representing the SQL query and submitting that to a database server object (we will discuss the ADO object model in more detail in section 6.1). So for instance, The query $\sigma_{President=False}\ Rogerson$ is expressed in Visual Basic as:

```
Q =     "SELECT *"
Q = Q & "FROM Rogerson AS r"
Q = Q & "WHERE r.President = FALSE"

Set RS = CreateObject("ADODB.Recordset")
RS.Open Q "Rogerson"

Do While Not RS.EOF
  Print RS("Object")
  Print RS("Edible")
  Print RS("Inheritance")
  Print RS("President")
  RS.MoveNext
Loop
```

3.3 The FP way: Comprehensions

Within the functional programming community, people have argued that (list) comprehensions are a good query notation for database programming languages [2]. For example using the comprehension notation supported by Haskell/DB, the query $\sigma_{President=False}\ Rogerson$ can be expressed as:

```
do{ r <- table rogerson
  ; restrict
```

```
      (r!president .==. constant False)
    ; return r
    }
```

The query $\pi_{Object}\ (\sigma_{Edible=True}\ Rogerson)$ becomes

```
  do{ r <- table rogerson
    ; restrict (r!edible .==. constant True)
    ; project (object = r!object)
    }
```

Queries that use projections and joins such as the following $\pi_{Name,Object}(Presidents \bowtie Rogerson)$ are harder to formulate because we have to indicate explicitly on which common fields to compare and how to create the resulting tuple:

```
  do{ r <- table rogerson
    ; p <- table president
    ; restrict (r!president .==. p!president)
    ; project (name = p!name,object = r!object)
    }
```

The comprehensions are fully *typed* and automatically translated into correct SQL strings which are send to a low-level database server. This paper describes not only how we did this for SQL but also tries to give a general recipe for embedding languages into a strongly typed language.

Let's put the question of embedding SQL aside until section 6 and first look how we in general can embed languages into Haskell.

4 Term embedding

Although SQL is embedded in this specific case, there is a general strategy for embedding services in a HOT language. We will illustrate this using a simple SQL expression service as an example. In SQL, expressions are used (amongst others) in the search conditions of `WHERE` clauses to perform computations and comparisons on columns and values.

4.1 SQL expression server

Lets assume that the SQL expression server provides us with the following interface for evaluating expressions (described in IDL):

```
interface IServer
  { void SetRequest([in,string] char* expr);
  ; void GetResponse([out] char* result);
  }
```

Although simple, the `IServer` interface captures the essence of many dynamic services such as a desk calculator, finger, HTTP, ftp, NNTP, DNS, ODBC, ADO and similar information servers.

From Haskell, we can access the `IServer` interface using the functions `setRequest`, and `getResponse` that are automatically generated by our *H/Direct* IDL compiler [6]:

```
setRequest :: String -> IServer s -> IO ()
getResponse :: IServer s -> IO String
```

We are now able to write an evaluator function that takes an expression, sends it to the server and returns the result:

```
runExpr :: String -> IO Int
runExpr = \expr ->
  do{ server <- createObject "Expr.Server"
    ; server # setRequest expr
    ; result <- server # getResponse
    ; return (read result)
    }
```

This is essentially the kind of interface that is in use now with SQL server protocols as ODBC or ADO. An unstructured SQL string is directly sent to the server. The problem is that there is nothing that prevents the programmer from sending invalid strings to the server, leading to errors at runtime and/or unpredictable behavior of the server. Clearly, this is unacceptable in critical business applications.

4.2 Abstract syntax

To prevent the construction of syntactically incorrect expressions, we define an *abstract syntax* for the terms of the input language of the specific server we are targeting, together with a "code generator" to map abstract syntax trees into the concrete syntax of the input language.

The abstract syntax `PrimExpr` simply defines literal constants, and unary and binary operators. (In section 6 we will add row selection):

```
data PrimExpr
  = BinExpr   BinOp PrimExpr PrimExpr
  | UnExpr    UnOp PrimExpr
  | ConstExpr String

data BinOp
  = OpEq | OpAnd | OpPlus | ...
```

Types `UnOp` and `BinOp` are just enumerations of the unary and binary operators of SQL expressions.

Writing expressions directly in abstract syntax is not very convenient, so we provide combinators to make the programmer's life more comfortable. Each SQL operator is represented in Haskell by the same operator surrounded with dots. Some definitions are[2]:

```
constant :: Show a => a -> PrimExpr
(.+.)    :: PrimExpr -> PrimExpr -> PrimExpr
(.==.)   :: PrimExpr -> PrimExpr -> PrimExpr
(.AND.)  :: PrimExpr -> PrimExpr -> PrimExpr
```

Now we can write `constant 3 .==. constant 5` instead of the cumbersome `BinExpr OpEq (ConstExpr (show 3)) (ConstExpr (show 5))`. This is what *embedded* domain specific languages are all about!

4.3 Concrete syntax

In order to evaluate expressions, we must map them into the exact *concrete* syntax that is required by the server component.

The code generator for our expression server is straightforward; we print expressions into their fully parenthesized concrete representation by a simple inductive function:

```
pPrimExpr :: PrimExpr -> String
pPrimExpr = \e ->
  case e of
   { ConstExpr s
       -> s
   ; UnExpr op x
       -> pUnOp op++parens x
   ; BinExpr op x y
       -> parens x++pBinOp op++parens y
   }

parens = \x -> "(" ++ pPrimExpr x ++ ")"
```

[2]The `constant` function is unsafe since any value in the `Show` class can be used. In the real library we introduce a separate class `ShowConstant` which is only defined on basic types.

Normally however, this step will be more involved as we will see in the SQL example.

4.4 Embedding expressions

Now that we know how to construct expressions and how to generate code from them, we can rewrite the evaluator function to use the structured expressions:

```
runExpr :: PrimExpr -> IO Int
runExpr = \expr ->
  do{ server <- CreateObject "Expr.Server"
    ; server # setRequest (pPrimExpr expr)
    ; result <- server # getResponse
    ; return (read result)
    }
```

We can now use SQL expressions in Haskell as if they were built-in. Function `runExpr` will dynamically compile a `PrimExpr` *program* into *target code*, execute that on the expression server and coerce the result back into a Haskell integer *value*:

```
sum :: Int -> PrimExpr
sum = \n ->
  if (n <= 0)
  then (constant 0)
  else (constant n .+. sum (n-1))

test = do{ runExpr (sum 10) }
```

5 Type embedding

The above embedding is already superior to constructing unstructured string to pass to the server because it is impossible to construct syntactically incorrect requests. However, it is still possible to construct *ill typed* request, as the following example shows:

```
do{ let wrong = (constant 3) .AND. (constant 5)
  ; result <- runExpr wrong
  ; print result
  }
```

Since the `PrimExpr` data type is completely untyped, we have no way to prevent the construction of terms such as `wrong` that might crash the server because the operands of the `AND` expression are not of boolean type.

5.1 Phantom types

We used abstract syntax trees to ensure that we can only generate syntactically correct request, but the billion dollar question of course is whether there is a similar trick to ensure that we can only generate type correct requests. Fortunately, the answer is yes! It is possible indeed to add an extra layer on top of `PrimExpr` that effectively serves as a type system for the input language of the expression server.

The trick is to introduce a new *polymorphic* type `Expr a` such that `expr :: Expr a` means that `expr` is an expression of type `a`. The type variable `a` in the definition of the `Expr` data type is only used to hold a type; it does not occur in the right hand side of its definition and is therefore never physically present:

```
data Expr a = Expr PrimExpr
```

Now we refine the types of the functions to construct values of type `Expr a` to encode the typing rules for expressions:

```
constant :: Show a => a -> Expr a
(.+.)  :: Expr Int -> Expr Int -> Expr Int
(.==.) :: Eq a=> Expr a-> Expr a-> Expr Bool
(.AND.):: Expr Bool -> Expr Bool-> Expr Bool
```

For example, the definition of `(.==.)` is now:

```
(.==.) :: Eq a=> Expr a-> Expr a-> Expr Bool
(Expr x) .==. (Expr y)
         = Expr (BinExpr OpEq x y)
```

By making the `Expr` type an abstract data type, we ensure that only the primitive functions can use the unsafe `PrimExpr` type. If we now use these combinators to write `(constant 3) .AND. (constant 5)`, the *Haskell type-checker* will complain that the type `Expr Int` of the operand `(constant 2)` does not match the required type `Expr Bool`.

The typing of expressions via phantom type variables extends immediately to values built using Haskell primitives. Our example function `sum` for instance, now has type `sum :: Int -> Expr Int`.

Phantom type variables have many other exciting uses, for instance in encoding inheritance and typing pointers [7]. Later we will show how we use multiple phantom type variables to give a type safe encoding of attribute selection in records.

6 Embedding SQL

Armed with the knowledge of how to safely embed a simple language into Haskell, we return to our original task of embedding SQL into Haskell.

6.1 The SQL server

We will use ActiveX Data Objects (ADO) as our SQL server component. ADO is a COM [11] framework that can use any ODBC compliant database; MS SQL Server, Oracle, DB/2, MS Access and many others. The ADO object model is very rich but we will use only a tiny fraction of its functionality.

ADO represents a relation as a `RecordSet` object. It creates a set of records from a query via its `Open` method:

```
dispinterface Recordset {
  void Open
    ([in,optional] VARIANT Source,
    ,[in,optional] VARIANT ActiveConnection,
    ,[in,optional] CursorTypeEnum CursorType,
    ,[in,optional] LockTypeEnum LockType,
    ,[in,optional] long Options
    );
  Bool    EOF();
  void    MoveNext();
  Fields* GetFields();
}
```

The first argument of the `Open` method is the source of the recordset, which can be an SQL string or the name of a table or a stored procedure. The second argument of the `Open` method can be a connection string, in which case a new connection is made to create the recordset, or it can be a `Connection` object that we have created earlier. In this paper, we will not use the other (optional) arguments of the `Open` method, hence we provide the following signature for `open`:

```
open :: (VARIANT src, VARIANT actConn) =>
  src -> actConnn -> IRecordSet r -> IO ()
```

The `MoveNext`, `EOF` and `GetFields` methods allow us to navigate through the recordset. Their Haskell signatures are:

```
moveNext  :: IRecordSet r -> IO ()
eof       :: IRecordSet r -> IO Bool
getFields :: IRecordSet r -> IO (IFields ())
```

The `Fields` interface gives access to all the fields of a row, they can be accessed either by position or by name:

```
dispinterface Fields {
    long   GetCount();
    Field* GetItem([in] VARIANT Index);
};
```

Each `Field` object corresponds to a column in the `Recordset`:

```
dispinterface Field {
  VARIANT GetValue();
  BSTR    GetName();
}
```

The `GetValue` property of a `Field` object can be used to obtain the value of a column in the current row. The `GetName` property returns the name of the field:

```
getValue :: VARIANT a => IField f -> IO a
getName  :: IField f -> IO String
```

Although the ADO object model is somewhat more refined than the expression evaluator example we have seen earlier, it does still fits the basic client-server framework. Requests are submitted via the `Open` method, and responses are inspected by iterating over the individual `Field` objects of the `Fields` collection.

6.2 Using the RecordSet in Haskell

In Haskell, we would like to abstract from iterating through the record set and access the result of performing a query as a list of fields. This faces us with the choice of either returning this list eagerly, or creating it lazily. In the former case, all fields are read into a list at once. In the latter case, the fields are encapsulated in a lazy stream where a field is read by demand.

Both functions are defined in terms of the function `readFields` that takes an `IO`-action transformer function as an additional argument:

```
readFields :: (IO a -> IO a)
           -> IRecordSet r -> IO [IFields ()]
readFields  = \perform -> \records -> perform $
  do{ atEOF <- records # eof
    ; if atEOF
      then do{ return [] }
      else do{ field <- records# etFields
             ; records#moveNext
             ; rest <- rs#readFields perform
             ; return ([field] ++ rest)
             }
    }
```

By taking `perform` to be the identity, we get a function that reads the list of fields strictly, by taking `perform` to be the IO-delaying function `unsafeInterleaveIO` we obtain a function that reads the list of fields lazily.

A simple query evaluator can now be written as:

```
runQuery :: String -> IO [IFields ()]
runQuery  = \query ->
  do{ records <- createObject "ADO.RecordSet"
    ; records # open query Nothing
    ; fields  <- records # readFields id
    ; return fields
    }
```

Of course, this approach suffers from all the weaknesses described in section 4.

6.3 Abstract syntax

Just as in the previous example we will define and *abstract syntax* for expressing database operations.

Our language for expressing those operations will be the relational algebra. The code generator will take these expressions and translate them to the concrete syntax of SQL statements which preserve the semantics of the original expression.

The abstract syntax for the relation algebra becomes[3]:

```
type TableName  = String
type Attribute  = String
type Scheme     = [Attribute]
type Assoc      = [(Attribute,PrimExpr)]

data PrimQuery
  = BaseTable TableName Scheme
  | Project   Assoc    PrimQuery
  | Restrict  PrimExpr PrimQuery
  | Binary    RelOp PrimQuery PrimQuery
  | Empty

data RelOp
  = Times | Union | Intersect
  | Divide | Difference

data PrimExpr
  = AttrExpr  Attribute
  | ConstExpr String
  | BinExpr   BinOp PrimExpr PrimExpr
  | UnExpr    UnOp PrimExpr
```

For example, the relational expression that returns all objects that are edible: $\pi_{Object}\ (\sigma_{Edible=True}\ Rogerson)$ can be expressed in our abstract syntax as:

```
Project [("Object",AttrExpr "Object")]
 (Restrict (BinExpr OpEq (AttrExpr "Edible")
                         (ConstExpr "True"))
   (BaseTable "Rogerson"
     [ "Object", "Edible"
     , "Inheritance", "President"
     ]
  ))
```

6.4 Concrete syntax

It is straightforward to generate concrete SQL statements from the `PrimQuery` data type, allthough special care has to be taken to preserve the correct semantics of the relational algebra due to the idiosyncrasies of SQL. The use of the relational algebra as an intermediate language allows us to target a wide range of different database languages. We are planning to add bindings to other dialects of SQL and languages as ASN.1.

Besides being portable, the simple semantics of the relational algebra allows us to perform a powerful set of optimizations quite easily before transforming the expression to concrete syntax. Many times the SQL server is not capable of doing these transformations due to the complex semantics of SQL. Another benefit is that we can add operations like table comparisons which are very hard to express in languages like SQL, but easy to generate from a relational expression.

6.5 Towards comprehensions

We could proceed as in our earlier example and define some friendly combinators for specifying relational expressions as we did in our previous example. However there is a serious drawback to using relational expressions directly as our programming language. In the relational algebra, attributes are only specified by their name. There is no separate binding mechanism to distinguish attributes from different tables. Suppose we take the cartesian product of a relation with itself. In SQL we could write:

```
SELECT X.Name, X.Mark
FROM Students As X, Students As Y
WHERE X.Mark = Y.Mark
  AND X.Name <> Y.Name
```

But in the relational algebra, we are unable to do this since there are common common attributes like `Name` and `City` which lead to ambiguity. To take the cartesian product, one relation needs to rename those attributes. The join ($\bowtie$) operator is especially introduced to make it easier to specify the most common products where renaming would be necessary. Besides only covering the most common cases, it is notoriously hard to typecheck join expressions [3] and we haven't found a way to embed those typing rules within Haskell.

However, why not use the same approach as SQL? We will introduce a binding mechanism (monad comprehensions) for qualifying relations. Instead of identifying attributes just by name, we will use both a name and relation. The above query is formulated in Haskell/DB as:

[3]The `Project` constructor actually does both projection and renaming.

```
do{ x <- table students
  ; y <- table students
  ; restrict (x!mark .==. y!mark)
  ; restrict (x!name .<>. y!name)
  ; project (name = x!name, grade = x!grade)
  }
```

Under the hood, we still generate relational algebra expressions but all the renaming is done automatically within the combinators. Besides automatic renaming, we would like the Haskell type-checker to prevent us from writing silly queries such as this one where we ask to project the non-existing `city` attribute of a student:

```
do{ x <- table students
  ; project (name = x!name, city = x!city)
  }
```

We will present two designs for implementing comprehensions that are increasingly more type safe, but at the same time increasingly complex.

6.6 Attempt 1: Untyped comprehensions

The first attempt only hides the automatic renaming of attribute names, making this solution already much safer and convenient than writing abstract syntax directly. In our next attempt we will use phantom types to make queries more type safe. We defined the `Query` monad to express our queries. The use of a monad gives us the following advantages:

- The `do` notation provides a nice syntax to write queries (comprehensions).
- Monads enable a custom binding mechanism (ie `do{x <- table X; ...}`). to qualify names. An alternative approach of explicitly renaming attributes would be too cumbersome to use in practice.
- An invisible state can be maintained. The state of the `Query` monad contains the (partially) completed relational expression and a fresh name supply for automatic renaming of attributes.

Our query language now consists of three basic combinators, `restrict`, `project` and `table`, and the two monad operations[4] `returnQ` and `bindQ` for the `Query` monad. Besides that we have the usual binary combinators like `union`:

```
type State = (PrimQuery,FreshNames)
data Query a = Query (State -> (a,State))

returnQ :: a -> Query a
bindQ :: Query a -> (a -> Query b) -> Query b

restrict :: Expr Bool -> Query ()
project :: Rec r -> Query Rel
table :: Table -> Query Rel

union :: Query Rel -> Query Rel -> Query Rel
```

The exact details of doing correct renaming for all attributes are rather subtle and a thorough discussion is outside the scope of this paper. We will provide all the details in a separate report [13]. The `Rec r` and `Rel` types are explained in the next section where we add typed layer on top of the comprehension language.

6.7 Attempt 2: Typed comprehensions

We already know how to make the expression sublanguage type safe using phantom types. For the comprehension language we will use the same trick. Central to this discussion is the attribute selection operator:

```
(!) :: Rel -> Attribute -> PrimExpr
```

Given a relation and an attribute name, the operator returns the attribute value expression. Given that any attribute always has a well defined type, we parametrize the attribute by its type to return an expression of the same type:

```
data Attr a   = Attr Attribute

(!) :: Rel -> Attr a -> Expr a
```

Although we can now only use an attribute expression at its right type, the system doesn't prevent us from selecting non-existent attributes from the relation. The solution is to parametrize the `Rel` type by the its "scheme". Similarly, we parametrize the `Attr` type again by both the scheme of the relation *and* the type of the attribute:

[4] These functions allow us to use the `do` syntax.

```
data Rel r    = Rel Scheme
data Table r  = Table TableName Scheme
data Attr r a = Attr Attribute
```

The `Rel` and `Table` both retain their associated scheme. This is needed to read the concrete values returned by the actual query.

The selection operator `(!)` now expresses in its type that given a relation with scheme `r` that has an attribute of type `a`, it returns an expression of type `a`. The polymorphic types of the other basic combinators should not be too surprising:

```
(!) :: Rel r -> Attr r a -> Expr a

restrict :: Expr Bool -> Query ()
project  :: Rec r -> Query (Rel r)
table    :: Table r -> Query (Rel r)
```

Our desire to guarantee type safety bears some additional cost on the user. For every attribute *attr* that occurs in a query, we have to define an attribute definition *attr* `::` `r\`*attr* `=>` `Attr (`*attr* `:: a | r) a` by hand, until TREX will provide first class labels. Similarly, for every base table with scheme *r* that we use, we need a definition of type `Table` *r*. For the example database of section 7 we have:

```
students :: Table(name :: String, mark :: Char)
name :: r\name =>
  Attr (name :: String | r) String
mark :: r\mark =>
  Attr (mark :: Char | r) Char
```

We have written a tool called DB/Direct that queries the system tables and automatically generates a suitable database definition. This tool is of course written using Haskell/DB.

The Haskell type-checker now checks the consistency of our queries. It accepts query `passed` without problems, but it fails to type check query `failed` because the condition `student!mark .<=. constant 5` wrongly attempts to compare a character to an integer and because the programmer accidentally used the attribute `ame` instead of `name`:

```
passed :: Query (Rel (name :: String))
passed =
  do{ student <- table students
    ; restrict (student!mark .>=. constant 'B')
    ; project (name = student!name)
    }

failed :: Query (Rel (name :: String))
failed =
  do{ student <- table student
    ; restrict (student!mark .<=. constant 5)
    ; project (name = student!ame)
    }
```

7 Exam marks

Any commercial exploitation of the web today uses server-side scripts that connect to a database and deliver an HTML page composed from dynamic data obtained from querying the database using information in the client's request. The following example is a is a simple server-side web script that generates an HTML page for a database of exam marks and student names.

The database is accessed via simple web page with a text entry and a submit button:

The underlying HTML has a form element that submits the query to the `getMark` script on the server.

```
<HTML>
<HEAD> <TITLE>Find my mark</TITLE> </HEAD>
<BODY>
  <FORM ACTION="getMark.asp" METHOD="post">
    My name is:
    <INPUT TYPE=text NAME="name">
    <INPUT TYPE="submit" VALUE="Show my mark">
  </FORM>
</BODY>
</HTML>
```

7.1 Visual Basic

Even the simplest Visual Basic solution uses no less than four different languages. Visual Basic for the business logic and glue, SQL for the query, and HTML with ASP directives to generate the result page.

1. In ASP pages, scripts are separated from the rest of the document by <% and %> tags. The prelude script declares all variables, construct the query and retrieves the results from the students database. The ASP `Request` object contains the information passed by the client to the server. The `Form` collection contains all the form-variables passed using a `POST` query. Hence `Request.Form("name")` returns the value that the user typed into the `name` textfield of the above HTML page.

```
<%
Q =     "SELECT student.name, student.mark"
Q = Q & " FROM Students AS student"
Q = Q & " WHERE "student.name = "
Q = Q & Request.Form("name")

Set RS = CreateObject("ADO.Recordset")
RS.Open Q "CS101"
%>
```

2. The body contains the actual HTML that is returned to the client, with a table containing the student's name and mark. The <%= and %> tags enclose Visual Basic expressions that are included in the output text. Thus the snippet:

```
<TR>
  <TD><%=RS("name")%></TD>
  <TD><%=RS("mark")%></TD>
</TR>
```

creates a table row that contains the name and the mark of the student who made the request:

```
<HTML>
<HEAD> <TITLE>Marks</TITLE> </HEAD>
<BODY>
  <TABLE BORDER="1">
    <TR>
      <TH>Name</TH>
      <TH>Mark</TH>
    <TR>
    <%Do While Not RS.EOF%>
    <TR>
      <TD><%=RS("name")%></TD>
      <TD><%=RS("mark")%></TD>
    <TR>
    <%RS.MoveNext%>
    <%Loop%>
  </TABLE>
</BODY>
</HTML>
```

3. The clean-up phase disconnects the databases and releases the recordset:

```
<%
RS.Close
set RS = Nothing
%>
```

7.2 Haskell

The Haskell version of our example web page is more coherent than the Visual Basic version. Instead of four different languages, we need only need Haskell embedded in a minimal ASP page [14]:

```
<%@ LANGUAGE=HaskellScript %>
<%
module Main where

import Asp
import HtmlWizard

main :: IO ()
main = wrapper $ \request ->
  do{ name <- request # lookup "name"
    ; r <- runQuery (queryMark name) "CS101"
    ; return (markPage r)
    }
```

The function `queryMark` is the analog of code in the prelude part of the Visual Basic page, except here it is defined as a separate function parametrized on the name of the student:

```
type Student = Row(name :: String, mark :: Char)

queryMark :: String -> Query Student
queryMark = \n ->
  do{ student <- table students
    ; restrict (student!name .==. lift n)
    ; project
        ( name = student!name
        , mark = student!mark
        )
    }
```

Function `markPage` makes a nice HTML page from the result of performing the query:

```
markPage :: [ Student ]
markPage = \rs ->
  page "Marks"
    [ table
        ( headers = [ "Name", "Mark"]
        , rows = [[r!name, r!mark:""]
                 | r <- rs
```

```
                  ]
              )
        ]
%>
```

The Haskell program is more concise and more modular than the Visual Basic version. Functions `queryMark` and `markPage` can be tested separately, and perhaps even more importantly, we can easily reuse the complete program to run in a traditional CGI-based environment, by importing the `CGI` module instead of `Asp` (in a language such as Standard ML we would have parametrized over the server interface).

8 Status and conclusions

The main lesson of this paper is a new design principle for embedding domain specific languages where embedded programs are compiled on-the-fly and executed by submitting the target code to a server component for execution. We have shown how to embed SQL into Haskell using this principle, but there are numerous other possible application domains where embedded compilers are the implementation technology of choice; many Unix services are accessible using a completely text-based protocol over sockets.

Traditionally, domain abstractions are available as external libraries. For instance, the JMAIL component (available at the time of writing at `http://www.dimac.net/`) provides a plethora of methods to compose email messages, to show just a few:

```
dispinterface ISMTPMail {
  VARIANT_BOOL Execute();
  void AddRecipient([in] BSTR Email);
  [propget] BSTR Sender();
  [propput] void Sender([in] BSTR rhs);
  [propget] BSTR Subject();
  [propput] void Subject([in] BSTR rhs);
  [propget] BSTR Body();
  [propput] void Body([in] BSTR rhs);
};
```

Instead of providing a whole bunch of methods to construct an email message in an imperative style, an alternative approach would be to have a raw (SMTP) mail server [16] component that accepts email messages in the RFC822 format [4] directly together with a set of combinators to build email messages in a compositional style. Ultimately, these abstract email messages are "compiled" into raw strings that are submitted to the mail server, perhaps by piping into the appropriate telnet port.

Our ultimate goal for a Domain Specific Embedded Compiler is to provide hard compile-time guarantees for type safety and syntactical correctness of the generated target program. Syntactical correctness of target programs can be garantueed by hiding the construction of programs behind abstract data types. Phantom types, polymorphic types whose type parameter is only used at compile-time but whose values never carry any value of the parameter type, are a very elegant mechanism to impose the Haskell type system on the embedded language.

Our final example shows how Domain Specific Embedded Compilers can make server-side web scripting more productive. Because we can leverage on the abstraction mechanisms of Haskell (higher-order functions, module system), compared to the VB solution, the Haskell program is of higher quality, and easier to change and maintain. The formulation of queries using the `do{}` notation and extensible records is rather neat, but the exact translation into SQL turned out to be rather subtle.

Both the Haskell/DB and the DB/Direct packages are available on the web at the URL `http://www.haskell.org/haskellDB`.

Acknowledgements

Thanks to Hans Philippi for brushing up our knowledge on databases, and to the DSL99 referees, Arjan van Yzendoorn and Jim Hook for their constructive remarks that helped to improve the presentation of our paper. Joe Armstrong's talk on services as components at the Dagstuhl workshop on "Component Based Development Under Different Paradigms" provided much of the initial inspiration for this work.

References

[1] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick II, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley Computer Publishing, 1998.

[2] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *ACM SIGMOD Record*, 23(1):87–96, March 1994.

[3] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, March 1996

[4] David H. Crocker. Standard for the Format of Arpa Internet Text Messages. Technical Report RFC 822, 1982. http://www.imc.org/rfc822.

[5] C.J. Date. *An Introduction to Database Systems (6th edition).* Addison-Wesley, 1995.

[6] S.O. Finne, D. Leijen, E. Meijer, and S.L. Peyton Jones. H/Direct: A Binary Foreign Language Interface for Haskell. In *ICFP'98*, 1998.

[7] S.O. Finne, D. Leijen, E. Meijer, and S.L. Peyton Jones. Calling hell from heaven and heaven from hell. In *ICFP'99*, 1999.

[8] B.R. Gaster and M.P. Jones. A Polymorphic Type System fo Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, 1996.

[9] Paul Hudak. Modular Domain Specific Languages and Tools. In *ICSR5*, 1998.

[10] Simon Peyton Jones and John Hughes (eds). Report on the Language Haskell'98. Available online: htpp://www.haskell.org/report, February 1999.

[11] S.L. Peyton Jones, E. Meijer, and D. Leijen. Scripting COM Components in Haskell. In *ICSR5*, 1998.

[12] P. J. Landin. The next 700 programming languages. *CACM*, 9(3):157–164, March 1966.

[13] D. Leijen and E. Meijer. Translating `do`-notation to SQL. 1999.

[14] Erik Meijer. Server Side Scripting in Haskell. *Journal of Functional Programming*, accepted for publication.

[15] S. L. Peyton Jones and Philip Wadler. Imperative functional programming. In *20'th ACM Symposium on Principles of Programming Languages*, Charlotte, North Carolina, January 1993.

[16] Jonathan B. Postel. Simple Mail Transfer Protocol. Technical Report RFC 821, 1982. http://www.imc.org/rfc821.

[17] Dale Rogerson. *Inside COM.* Microsoft Press, 1997.