The Power of Pi

Nicolas Oury Wouter Swierstra
University of Nottingham
{npo,wss}@cs.nott.ac.uk

Abstract

This paper exhibits the power of programming with dependent types by dint of embedding three domain-specific languages: Cryptol, a language for cryptographic protocols; a small data description language; and relational algebra. Each example demonstrates particular design patterns inherent to dependently-typed programming. Documenting these techniques paves the way for further research in domain-specific embedded type systems.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Theory

1. Introduction

Dependent types matter. And not just for program verification and proof assistants: dependent types matter to programmers. Whether you want to interface to a database, write a webserver, or manipulate binary data, dependent types can make a difference.

This paper demonstrates *how to program* with dependent types. In particular, we present three case studies. Each case study describes a domain-specific language that is difficult to embed in conventional functional languages such as Haskell (Peyton Jones 2003):

- Wadler (1987) has recognised the importance of defining custom pattern matching principles for inductive data types. Such *views*, when implemented in a language with dependent types, carefully maintain the relation between the data being eliminated and the custom patterns. We illustrate these techniques by showing how to embed Cryptol (Galois, Inc. 2002), a high-level specification language for low-level cryptographic algorithms, in a dependently typed language (Section 2). In particular, we will write a domain-specific library for Cryptol's most characteristic feature: bitvectors of a fixed length equipped with a special pattern matching principle.
- To facilitate processing data written in custom file formats, there has been a great deal of research on data description languages (Back 2002; McCann and Chandra 2000; Fisher and Gruber 2005). Given a file format description, these languages generate data types that represent the data stored in such custom formats, together with a parser. Using dependent types, we will write a *universe* capturing a large collection of file formats, together with a generic parser and pretty printer for any file format that can be described in this universe (Section 3).
- Despite numerous efforts (Leijen and Meijer 1999; Bringert et al. 2004), Haskell does not have an elegant, strongly-typed database interface. Existing libraries strike an awkward balance between encoding static invariants using type classes and resorting to dynamic type checking. We will write a type-safe combinator library for database queries using dependent types that does not require any kind of preprocessor or external tool (Section 4).

We will *not* try to give a complete embedding of each of these three domain-specific languages. Each of the above examples serves to introduce new concepts: the embedding itself is a means, not an end. Despite this limitation, we make several novel contributions:

- These examples document some of the emerging design patterns of dependently-typed programming. This is an incipient field of research with only a handful of recognised specialists. Some of the design patterns, such as views and universes, have been part of the community's folklore for some time. This is the first time they have been presented in a single, uniform fashion with concrete, real-world examples that illustrate their importance.
- Our case studies show how domain-specific languages are an important application domain of dependently-typed programming. Besides embedding the *terms* of a domain-specific language, we show how to enforce invariants statically. Indeed, we show how to write *domain-specific embedded type systems*. By embedding such type systems we inherit all the meta-theoretical properties, such as decidable type checking or subject reduction, of the host language. We believe this is a particularly exciting new field of research, that enables the rapid prototyping of type systems.
- Finally, we have tried to identify the key benefits that programming with dependent types afford (Section 5). Our examples already illustrate some of the advantages that dependently-typed programming languages hold over mainstream functional languages. We hope that making these advantages explicit will not only help direct today's research in the design of tomorrow's functional languages, but also enable programmers to use dependent types to greater effect.

Throughout this paper, we will use the latest installment of the dependently-typed programming language Agda (Norell 2007; The Agda Team 2008) as a vehicle of explanation. The code we present has been type-checked and can be executed on the computer sitting on your desktop.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

2. Domain-specific Embedded Cryptol

Cryptol (Galois, Inc. 2002) is a domain-specific language for cryptographic protocols developed by Galois, Inc. in consultation with cryptographers from the National Security Agency. It provides developers with a high-level, declarative specification language for low-level cryptographic algorithms. Such cryptographic algorithms perform various manipulations on sequences of bits. From a programming language designer's perspective, Cryptol has two distinguishing characteristics to facilitate these manipulations:

- The length of a sequence of bits is recorded in its *type*. This makes it impossible to mistake a 32-bit word for a 64-bit word, for example. Such sequences may be written using binary, decimal, octal, or hexadecimal constants. In Cryptol, we could define a number x as follows:

  $$x : [8];$$
  $$x = 42;$$

  The constant x is an 8-bit word, written $[8]$, defined to be equal to (the binary representation of) 42.

- Crypto-algorithms often split a word into pieces, manipulate the pieces, and then join them together again. Cryptol has special pattern-matching support that makes it easier to write such functions. For example, the *swab* function in Cryptol swaps the first and second byte of a 32-bit word:

  $$swab : [32] \to [32];$$
  $$swab\ [a\ b\ c\ d] = [b\ a\ c\ d];$$

  Note that the length of the pattern $[a\ b\ c\ d]$ determines the size of its constituent variables. We could also have written the pattern $[a\ b]$, and both a and b would be 16-bit words.

These are Cryptol's two most distinguishing characteristics. Apart from these features, Cryptol consists of a simple, pure, first-order functional language with various kinds of syntactic sugar, such as list comprehensions and a special notation for polynomials.

It is important to emphasise that Cryptol is *not* a domain-specific *embedded* language. Galois, Inc. have specifically developed a Cryptol interpreter and compiler—a task that certainly entails considerable development effort. To embed Cryptol in any other language, the principal challenge we must face is facilitating Cryptol's pattern matching on bit vectors. Before we tackle this problem, we need to describe the syntax of Agda.

2.1 Introducing Agda

In Agda, new data types can be defined using a syntax similar to Haskell's syntax for GADTs (Peyton Jones et al. 2006). For example, we could define the natural numbers as follows:

$$\begin{array}{l}\mathbf{data}\ \mathit{Nat} : \mathit{Set}\ \mathbf{where}\\ \quad \mathit{Zero} : \mathit{Nat}\\ \quad \mathit{Succ} : \mathit{Nat} \to \mathit{Nat}\end{array}$$

The Nat data type has two constructors: $Zero$ and $Succ$. Note that we must explicitly give the type of the Nat data type. In this instance, $Nat : Set$ means that Nat is a base type. Programmers familiar with Haskell may want to think of the Agda type Set as $\star$, the kind of all base types. Agda allows us to use integer literals to write these numbers.

We can define functions over natural numbers using pattern matching and recursion. For instance, we could define addition as follows:

$$\begin{array}{l}\_+\_ : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\ \mathit{Zero}\ \ \ \ + n = n\\ (\mathit{Succ}\ k) + n = \mathit{Succ}\ (k + n)\end{array}$$

Agda uses underscores to denote the positions of arguments when defining new operators. Using this notation, it is possible to define infix, postfix, or even mixfix operators.

Polymorphic lists are a little bit more interesting than natural numbers.

$$\begin{array}{l}\mathbf{data}\ \mathit{List}\ (A : \mathit{Set}) : \mathit{Set}\ \mathbf{where}\\ \quad \mathit{Nil} : \mathit{List}\ A\\ \quad \mathit{Cons} : A \to \mathit{List}\ A \to \mathit{List}\ A\end{array}$$

We can parameterise a data type by listing its arguments immediately after the data types name. In this example, we have parameterised the type of lists by a type variable A of type Set.

Just as we added natural numbers, we can append two lists:

$$\begin{array}{l}\mathit{append} : (A : \mathit{Set}) \to \mathit{List}\ A \to \mathit{List}\ A \to \mathit{List}\ A\\ \mathit{append}\ A\ \mathit{Nil}\ ys\ \ \ \ \ \ \ \ \ \ \ = ys\\ \mathit{append}\ A\ (\mathit{Cons}\ x\ xs)\ ys = \mathit{Cons}\ x\ (\mathit{append}\ A\ xs\ ys)\end{array}$$

Note that the type of *append* uses the *dependent function space*, or Π-type, written $(x : \sigma) \to \tau$, where the variable x may occur in the type τ. The crucial difference from the usual function space, as is present in Haskell, is its elimination rule:

$$\frac{\Gamma \vdash e_1 : (x : \sigma) \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : \tau[x \mapsto e_2]}$$

The resulting *type* of the application $e_1\, e_2$ now *depends* on the *value* of the argument e_2. The use of the dependent function space in the *append* function is not very interesting: it corresponds to parametric polymorphism.

It can be quite tedious to instantiate type variables, such as A in the *append* function, by hand. Agda allows you to mark certain arguments as *implicit* by enclosing them in curly braces. Agda will automatically instantiate these arguments, much in the same way a Haskell compiler automatically instantiates type variables. Using implicit arguments, our *append* function becomes:

$$\begin{array}{l}\mathit{append} : \{A : \mathit{Set}\} \to \mathit{List}\ A \to \mathit{List}\ A \to \mathit{List}\ A\\ \mathit{append}\ \mathit{Nil}\ ys\ \ \ \ \ \ \ \ \ \ \ = ys\\ \mathit{append}\ (\mathit{Cons}\ x\ xs)\ ys = \mathit{Cons}\ x\ (\mathit{append}\ xs\ ys)\end{array}$$

Besides parameterising data types, we can also *index* data types by values. The classic example of an indexed family is the type of vectors:

$$\begin{array}{l}\mathbf{data}\ \mathit{Vec}\ (A : \mathit{Set}) : \mathit{Nat} \to \mathit{Set}\ \mathbf{where}\\ \quad \mathit{Nil} : \mathit{Vec}\ A\ \mathit{Zero}\\ \quad \_::\_ : \{n : \mathit{Nat}\} \to A \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ (\mathit{Succ}\ n)\end{array}$$

The Vec type takes two arguments: a type and a number. The Vec type is indexed by this number, but parameterised by the type A. Like GADTs in Haskell, the codomains of the constructors of Vec are different. Unlike GADTs, however, Agda's *indexed families* cannot only be indexed by *types* but may also be indexed by *values*.

Just as we declared infix operators, we can declare infix constructors by writing underscores to denote the positions of arguments. Note that Agda allows us to use the same constructor name, Nil, for different data types. We will be asked to add an explicit type signature if there is any ambiguity.

Once again, we can define an operation to append two vectors:

$$\begin{array}{l}\_+\!\!+\_ : \mathbf{forall}\ \{A\ m\ n\} \to\\ \quad \mathit{Vec}\ A\ m \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ (m + n)\\ \mathit{Nil}\ \ \ \ \ \ +\!\!+ ys = ys\\ (x :: xs) +\!\!+ ys = x :: (xs +\!\!+ ys)\end{array}$$

Here we use the keyword **forall** to group together several implicit arguments and omit their types. We will occasionally use the forall-notation to make our type signatures easier to read. Note that there

is no restriction on the type of implicit arguments: we have made n and m implicit arguments, despite being numbers and not types.

When programming with dependent types the type checker sometimes needs to perform evaluation: it would be rather unfortunate if $Vec\ A\ 3$ and $Vec\ A\ (1 + 2)$ were distinct types. To ensure type checking remains decidable, Agda requires you to write total functions, that is, your function definitions must be obviously structurally recursive and cover all case alternatives. If you fail to do so, Agda warns you that your definition is dubious and may cause the type checker to loop.

2.2 Embedding Cryptol's Types

We can now begin to embed Cryptol in Agda. A binary word is simply a vector of bits:

$$\begin{array}{l}\mathbf{data}\ Bit : Set\ \mathbf{where}\\ \quad O : Bit\\ \quad I\ : Bit\\ Word : Nat \to Set\\ Word\ n = Vec\ Bit\ n\end{array}$$

Introducing Cryptol's pattern matching principle on a $Word$ requires a bit more effort. Before we define such a view, we need to understand how pattern matching works in the presence of dependent types.

2.3 Pattern matching in the presence of dependent types

With the introduction of GADTs, pattern matching in Haskell has become more subtle. Consider the following Haskell data type:

$$\begin{array}{l}\mathbf{data}\ Expr\ a\ \mathbf{where}\\ \quad B : Bool \to Expr\ Bool\\ \quad I\ : Int \to Expr\ Int\end{array}$$

We can write a simple evaluation function for this mini-language:

$$\begin{array}{l}eval :: Expr\ a \to a\\ eval\ e = \mathbf{case}\ e\ \mathbf{of}\\ \quad B\ b \to b\\ \quad I\ x\ \to x\end{array}$$

The two case branches have different types. This is quite surprising: pattern matching may suddenly introduce equations between *types*. In the presence of dependent types, pattern matching may introduce equations between *values*, as we will show next.

Suppose we define the following data type:

$$\begin{array}{l}\mathbf{data}\ \_\equiv\_\ \{A : Set\} : A \to A \to Set\ \mathbf{where}\\ \quad Refl : \{x : A\} \to x \equiv x\end{array}$$

A value of type $(x \equiv y)$ corresponds to a proof that x and y are equal. The $\equiv$-type is parameterised by an implicit type A and indexed by two values of that type. It has a single constructor, $Refl$, that corresponds to a proof that any x is equal to itself. This type plays a fundamental role in type theory (Nordström et al. 1990).

Whenever we pattern match on such a proof, we learn how two values are related. For example, suppose we want to write the following function:

$$f : (x : Nat) \to (y : Nat) \to (p : x \equiv y) \to Nat$$

What patterns should we write for x, y and p? Clearly, p must be $Refl$. As soon as we match on p, however, we learn something about x and y, i.e., they must be the same. Throughout this paper, we will write this as follows:

$$f\ x\ \lfloor x \rfloor\ Refl = ...$$

The pattern $\lfloor x \rfloor$ means 'the value of this argument can only be equal to x.' In Agda you must currently write out how the different patterns relate by hand; Epigram (McBride and McKinna 2004), on the other hand, demonstrated that this process can be automated. Occasionally, we may not be interested in the information we learn, in which case we will use the underscore as a wildcard pattern:

$$f\ x\ \_\ Refl = ...$$

While not strictly necessary, we believe that writing out patterns such as $\lfloor x \rfloor$ explicitly serves as important, machine-checked documentation of what we learn from pattern matching.

2.4 Views

As the last section shows, dependent types change the way we must think about pattern matching. Unfortunately, we are still no closer to defining Cryptol's special pattern matching principle on vectors. Before we do so, we will cover a simpler example. The techniques we present are not new (McBride and McKinna 2004), although their application is.

The traditional definition of lists that we gave in Section 2.1 is biased: it's easy to recurse over the list starting from the front. Recursing in the other direction, back to front, is a bit awkward. Let us address this imbalance by defining a 'snoc-view' that allows programmers to recurse over a list, starting with the last element.

We begin by defining the $SnocView$ data type:

$$\begin{array}{l}\mathbf{data}\ SnocView\ \{A : Set\} : List\ A \to Set\ \mathbf{where}\\ \quad Nil\ \ : SnocView\ Nil\\ \quad Snoc : (xs : List\ A) \to (x : A) \to\\ \qquad SnocView\ (append\ xs\ (Cons\ x\ Nil))\end{array}$$

This $SnocView$ type is indexed by a list. The constructors correspond to the patterns, $Snoc$ and Nil, that we want to pattern match. The index itself relates these patterns to a 'regular list.' The pattern Nil corresponds to the empty list; the pattern $Snoc\ xs\ x$ corresponds to the list $append\ xs\ (Cons\ x\ Nil)$.

Next, we need to define a function $view$ that takes any list xs and produces a value of type $SnocView\ xs$.

$$\begin{array}{ll}view : \{A : Set\} \to (xs : List\ A) \to SnocView\ xs & \\ view\ Nil = Nil & \\ view\ (Cons\ x\ xs) & \mathbf{with}\ view\ xs\\ view\ (Cons\ x\ \lfloor Nil \rfloor) & \mid Nil\\ \quad = Snoc\ Nil\ x & \\ view\ (Cons\ x\ \lfloor append\ ys\ (Cons\ y\ Nil) \rfloor) & \mid Snoc\ ys\ y\\ \quad = Snoc\ (Cons\ x\ ys)\ y & \end{array}$$

The case for the empty list is easy. If the list is non-empty, we need to pattern match on a recursive call to $view\ xs$. Agda's notation for case-expressions will probably look a bit unfamiliar.

In the previous section, we saw how pattern matching can introduce equalities between a function's arguments. Suppose we were to write case-expressions using the syntax of Haskell or ML:

$$\begin{array}{l}view\ (Cons\ x\ xs) =\\ \quad \mathbf{case}\ (view\ xs)\ \mathbf{of}\\ \qquad (Nil) \qquad \to ...\\ \qquad (Snoc\ ys\ y) \to ...\end{array}$$

When we pattern match on the call to $view\ xs$, however, we learn something about xs: in the first branch we know that xs is empty; in the second branch, xs is constructed by adding an element y to the end of a list ys. In the case expression above, however, this new information is not made apparent in the pattern $Cons\ x\ xs$. Therefore, it makes sense to repeat *all* a function's arguments for *every* branch of a case-expression. Agda's **with**-rule (McBride and McKinna 2004; Norell 2007) does precisely this.

After the left-hand side of a function definition, the keyword **with** marks the beginning of another expression on which you

want to pattern match. The subsequent clauses defining the function repeat the entire left-hand side of the function, followed by a vertical bar that marks the beginning of the new pattern. In the *view* function, for example, we pattern match on the recursive call $view\ xs$ with the patterns Nil and $Snoc\ ys\ y$. In both cases, we learn what xs must be: if $view\ xs$ is Nil, then xs is empty; if $view\ xs$ is $Snoc\ ys\ y$, then xs is equal to $append\ ys\ (Cons\ y\ Nil)$.

Once you understand the syntax of the **with**-rule, the definition of the *view* function should be straightforward. In the non-empty case, $Cons\ x\ xs$, we check whether or not x is the last element. If x is the last element, we return $Snoc\ Nil\ x$; if the rest of the list xs is not empty, we know it is built by adding an element y to the end of the prefix ys, and return $Snoc\ (Cons\ x\ ys)\ y$ accordingly.

To view a list backwards, we simply call the *view* function and pattern match on the result. For example, we may want to rotate a list one step to the right, removing the last element and adding it to the front:

$$
\begin{aligned}
&rotateRight : \{A : Set\} \to List\ A \to List\ A\\
&rotateRight\ xs\ \mathbf{with}\ view\ xs\\
&rotateRight\ \lfloor Nil \rfloor \qquad\qquad\qquad\qquad\quad \mid Nil\\
&\quad = Nil\\
&rotateRight\ \lfloor append\ ys\ (Cons\ y\ Nil) \rfloor \mid Snoc\ ys\ y\\
&\quad = Cons\ y\ ys
\end{aligned}
$$

It is important to emphasise that the patterns $\lfloor Nil \rfloor$ and $\lfloor append\ ys\ (Cons\ y\ Nil) \rfloor$ can be inferred automatically. Alternatively, we could have replaced them by underscores; the only part the user is responsible for writing is $view\ xs$ and the right-hand sides of the equations. Using the Agda interpreter, we can now check that our function behaves as we would expect:

```
Main> rotateRight (Cons 1 (Cons 2 (Cons 3 Nil)))
Cons 3 (Cons 1 (Cons 2 Nil))
```

This example shows how to write a custom pattern matching principle, or *view*, in a dependently-typed language. Wadler (1987) has presented numerous examples of views that can all be implemented in this fashion.

2.5 Cryptol's view

The code in Figure 1 implements Cryptol's special view on bitvectors. Before we define the view itself, we need a few simple auxiliary functions: $take$, $drop$, $split$, and $concat$. As before, defining the view entails two steps: defining a data type $SplitView$ indexed by a vector; and defining a function $view$ that takes any vector xs to a value of type $SplitView\ xs$.

The $SplitView$ data type is indexed by a vector of length $n \times m$. It has a single constructor corresponding to the pattern we wish to match, i.e., a vector of vectors. We try to mimic Cryptol's syntax by defining this constructor as a pair of square brackets surrounding the pattern.

Defining the $view$ function is a bit tricky. We would like to $split$ the argument vector into m pieces of size n, and pass these pieces to the $[\_]$ constructor:

$$view\ n\ m\ xs = [split\ n\ m\ xs]$$

There is a problem with this simple definition. It will return a value of type $SplitView\ m\ (concat\ (split\ n\ m\ xs))$ and not one of type $SplitView\ m\ xs$. How can we be so sure that $concat\ (split\ xs)$ is the same as the original vector xs? How can we convince the type-checker of this fact?

The solution is simple. We need to prove a lemma:

$$
\begin{aligned}
&splitConcatLemma : \mathbf{forall}\ \{A\ n\ m\} \to\\
&\quad (xs : Vec\ A\ (m \times n)) \to concat\ (split\ n\ m\ xs) \equiv xs
\end{aligned}
$$

As we showed in Section 2.3, when we pattern match on a proof, we introduce equations between values. In the same way that a Haskell type checker uses equations between types to check the branches of the $eval$ function in Section 2.3, Agda's type checker uses the equations between two values during type checking. The code in Figure 1 therefore matches on $concat\ (split\ n\ m\ xs)$, the proof that this is equal to xs, and the right-hand side we would like to write, $[split\ n\ m\ xs]$. When we then return $[split\ n\ m\ xs]$, we can be sure that it has the right type. The syntax for pattern matching on more than one intermediate value separates the values and patterns that you wish to match on by vertical bars, both after the **with** and on the left-hand side of the function definition.

The proof of $splitConcatLemma$ is not terribly interesting. We perform induction on m and require a lemma about $take$ and $drop$. The entire proof is about ten lines long, but may be quite hard to understand for readers unfamiliar with type theory. We refer to existing literature for a more thorough treatment of how to write such proofs (Nordström et al. 1990).

It is quite important to emphasise that such proofs are only ever visible to the view's implementor. Any user who wants to call the $view$ function never, ever writes a proof. We have just done all the hard work for them. In fact, we can even avoid proving this lemma altogether by defining an intermediate view:

$$
\begin{aligned}
&\mathbf{data}\ TakeView\ (A : Set)\ (m : Nat)\ (n : Nat)\\
&\quad : Vec\ A\ (m + n) \to Set\ \mathbf{where}\\
&\qquad Take : (xs : Vec\ A\ m) \to (ys : Vec\ A\ n)\\
&\qquad\qquad \to TakeView\ A\ m\ n\ (xs +\!\!+ ys)
\end{aligned}
$$

We can iteratively apply this view to split a vector into its constituent parts.

Finally, we can use the $view$ function to write Cryptol's $swab$ function in Agda:

$$
\begin{aligned}
&swab : Word\ 32 \to Word\ 32\\
&swab\ xs\ \mathbf{with}\ view\ 8\ 4\ xs\\
&swab\ \lfloor \_ \rfloor \mid [a :: b :: c :: d :: Nil]\\
&\quad = concat\ (b :: a :: c :: d :: Nil)
\end{aligned}
$$

2.6 Discussion

There are many other features of the Cryptol *toolkit*, such as compilation to C or FPGAs, that we have not discussed. Providing a full implementation of all the technical features that Galois's Cryptol compiler supports will still require a lot of work. We feel, however, that we have managed to capture the essence of the Cryptol *language*. The embedding we have presented here opens the door for several new exciting directions for further research, in particular, the formal verification of Cryptol algorithms using Agda.

One difference between our view and Cryptol's pattern matching principle is that we must explicitly pass natural numbers to our view function. Although we have managed to reduce this overhead slightly and have a single number suffice, this significantly complicates the code. The underlying problem is that Agda does not use information about the patterns we write to instantiate implicit arguments.

Could we have written this in Haskell? One might be tempted to write the snoc-view as follows:

$$
\begin{aligned}
&\mathbf{data}\ SnocView\ a = Nil \mid Snoc\ (SnocView\ a)\ a\\
&view :: [a] \to SnocView\ a
\end{aligned}
$$

Yet there are two important limitations of this version. First of all, when we view a list xs backwards, the connection between $view\ xs$ and the original list is lost. In the dependently-typed view we presented previously, we are explicit about what we learn about the original list when pattern matching on the view.

```
take : forall { A m } → (n : Nat) → Vec A (n + m) → Vec A n
take Zero     l        = Nil
take (Succ k) (x :: xs) = x :: take k xs

drop : forall { A m } → (n : Nat) → Vec A (n + m) → Vec A m
drop Zero     xs       = xs
drop (Succ k) (x :: xs) = drop k xs

split : forall{ A } → (n : Nat) → (m : Nat) → Vec A (m × n) → Vec (Vec A n) m
split n Zero     Nil    = Nil
split n (Succ k) xs     = (take n xs) :: (split n k (drop n xs))

concat : forall { A n m } → Vec (Vec A n) m → Vec A (m × n)
concat Nil         = Nil
concat (xs :: xss) = xs ++ concat xss

data SplitView { A : Set } : { n : Nat } → (m : Nat) → Vec A (m × n) → Set where
  [_] : forall { m n } → (xss : Vec (Vec A n) m) → SplitView m (concat xss)

view : { A : Set } → (n : Nat) → (m : Nat) → (xs : Vec A (m × n)) → SplitView m xs
view n m xs with concat (split n m xs) | [split n m xs] | splitConcatLemma m xs
view n m xs | ⌊xs⌋ | v | Refl = v
```

Figure 1. Cryptol's view

The second problem is that the type of the *view* function in Haskell is too general. For example, the *view* function could be the constant function that always returns *Nil*. Originally, Wadler suggested that such view functions should always form one part of an isomorphism. Clearly the type of the Haskell view function provides no such guarantee.

The type of the view function we defined on Agda, on the other hand gives us much more information:

```
view : { A : Set } → (xs : List A) → SnocView xs
```

In general, we can always define a left-inverse for any view defined in this style by induction over the data type that the view returns. This is a more liberal condition on views than the isomorphism proposed by Wadler that more accurately reflects what views are about: you can view data any way you want, provided you never throw information away.

Epigram takes these ideas one step further and implements a clever *elaboration mechanism* that provides special programming support for using complex views.

3. Embedded data description languages

Programs manipulate data. Unfortunately, not all data conforms to a standard format. Crash reports, webserver logs, financial statistics, student marksheets, or billing information are all examples of the kind of data that may be represented by non-standard in-house formats. As a result, there are not always off-the-shelf libraries available to manipulate such data. Developers must waste time writing parsers or data conversion scripts.

To combat this problem, there is ongoing research into *data description languages* such as PADS (Fisher and Gruber 2005; Fisher et al. 2006), Packet Types (McCann and Chandra 2000), and Data Script (Back 2002). Essentially, such systems provide a *domain-specific language* that can be used to give a precise description of a data format. A separate tool then takes a data description file and produces a parser for that data format, together with a data type that represents the values inhabiting the data format.

Yet these are *external tools* that generate certain parts of your program from your data description file. Before programmers can use such tools, however, they must learn a separate language. Furthermore, the domain-specific data description language may not support all the abstractions of a general purpose programming language.

In this section we will implement a tiny data description combinator library in Agda, inspired by the Data Description Calculus (Fisher et al. 2006). There is no preprocessor or external tool involved: programmers may specify a file format using all the abstractions Agda has to offer.

3.1 Universes

Before we develop our combinator library, however, we need another type theoretic intermezzo. *Universes* are a fundamental concept in type theory. We explain what a universe is using a concrete example that should be familiar to Haskell programmers.

Agda does not have type classes. Yet our years of experience with Haskell has underlined the importance of ad hoc polymorphism. How might we achieve the same in a dependently-typed programming language?

Type classes are used to describe the collection of types that support certain operations, such as a decidable equality. The same issue also arises in type theory, where you may be interested in a certain collection of types that share some property, such as having a finite number of inhabitants. It is unsurprising that the techniques from type theory for describing such collections of types can be used to implement type classes.

Consider the following type U:

```
data U : Set where
  BIT : U
  CHAR : U
  NAT : U
  VEC : U → Nat → U
```

The data type U contains 'codes' for types: every data constructor of U corresponds to a type. In a dependently-typed language, we can define the decoding function *el* as follows:

```
el : U → Set
el BIT       = Bit
el CHAR      = Char
el NAT       = Nat
el (VEC u n) = Vec (el u) n
```

The pair of a type U and a function $el : U \to Set$ is called a *universe*. We can now define operations on the types in this universe by induction on U. For example, every type represented by U can be rendered as a $String$:

```
show : {u : U} → el u → String
show {BIT} O          = "O"
show {BIT} I          = "I"
show {CHAR} c         = charToString c
show {NAT} Zero       = "Zero"
show {NAT} (Succ k)   = "Succ " ++ parens (show k)
show {VEC u Zero} Nil = "Nil"
show {VEC u (Succ k)} (x :: xs)
  = parens (show x) ++ " :: " ++ parens (show xs)
parens : String → String
parens str = "(" ++ str ++ ")"
```

Note that we can pattern match on an implicit argument by enclosing a pattern in curly brackets. For any pattern p that matches an explicit argument, the corresponding pattern $\{p\}$ will match on its implicit counterpart.

This definition *overloads* the *show* function. When we call *show*, Agda will fill in the implicit argument of type U for us, allowing us to call *show* on arguments with different types:

```
Main> show I ++ " is binary for " ++ show 1
"I is binary for Succ (Zero)"
```

Note that, in contrast to Haskell's type classes, the data type U is closed. We cannot add new types to the universe without extending the data type U and the function el.

Clearly, *show* is not the only operation that the types represented by U have in common. In particular, we will later need a function that tries to parse a value of type U given a list of bits:

```
read : (u : U) → List Bit → Maybe (el u, List Bit)
```

We have omitted the definition of *read* as it is unremarkable.

3.2 The file format universe

The heart of our combinator library is formed by the *Format* data type below. Every value of type *Format* specifies a data file format. The *Bad* and *End* constructors correspond to failure and success respectively. The format *Base u* describes a data file consisting exclusively of a single value of type *el u*. The *Plus* constructor introduces left-biased choice. A parser for *Plus* f_1 f_2 will try to parse the format determined by f_1. Only when that fails, will it try to parse f_2.

Finally, there are two ways to sequence formats: *Skip* and *Read*. The *Skip* constructor sequences two formats, discarding any information stored in the first format. The *Read* constructor, on the other hand, sequences two formats, recording the information stored in both its arguments.

Many data file formats consist of a header, describing the contents of the file, followed by the actual data. The header often contains information about the *type* of the rest of the data stored in the file. Therefore, it is not enough for the *Read* constructor to construct a new *Format* from two arguments of type *Format*: the type of the second format may *depend* on the result of reading in the header. To capture this dependency, we need to parameterise the second argument of the *Read* constructor by the data type corresponding to the first argument.

This motivates the following mutually recursive definition of the *Format* data type together with the function $[\![\_]\!] : Format \to Set$, that calculates the data type resulting from parsing a given file format.

```
data Format : Set where
  Bad  : Format
  End  : Format
  Base : U → Format
  Plus : Format → Format → Format
  Skip : Format → Format → Format
  Read : (f : Format) → ([[f]] → Format) → Format
[[_]] : Format → Set
[[Bad]]       = Empty
[[End]]       = Unit
[[Base u]]    = el u
[[Plus f1 f2]] = Either [[f1]] [[f2]]
[[Read f1 f2]] = Sigma [[f1]] (λx → [[f2 x]])
[[Skip _ f]]  = [[f]]
data Sigma (A : Set) (B : A → Set) : Set where
  Pair : (x : A) → B x → Sigma A B
data Either (A : Set) (B : Set) : Set where
  Inl : A → Either A B
  Inr : B → Either A B
```

The *Empty* and *Unit* types in the definition above correspond to data types with zero and one inhabitant respectively. The result of *Read* f_1 f_2 is *not* simply a pair of $[\![f_1]\!]$ and $[\![f_2]\!]$. To reflect the potential dependency between f_1 and f_2 we need a *dependent pair*, or Σ-type, where the type of the second component may depend on the value of the first.

The *Format* data type and $[\![\_]\!]$ function form another universe: the universe of file formats.

3.3 Format combinators

We can now define several file format combinators, much in the style of the monadic parser combinators (Hutton and Meijer 1998). The simplest combinator consists of a file containing a single character:

```
char : Char → Format
char c = Read (Base CHAR)
              (λc' → if c ≡ c' then End else Bad)
```

Of course, we can be a bit more general and introduce a combinator that abstracts over which predicate must be satisfied:

```
satisfy : (f : Format) → ([[f]] → Bool) → Format
satisfy f pred =
  Read f (λx → if (pred x) then End else Bad)
```

There are two combinators to sequence formats, corresponding directly to *Skip* and *Read*. We have chosen to give them suggestive names:

```
_ ≫ _ : Format → Format → Format
f1 ≫ f2 = Skip f1 f2
_ ≫= _ : (f : Format) → ([[f]] → Format) → Format
x ≫= f = Read x f
```

Using these combinators, we can already define a simple file format. NETPBM is a collection of graphics programs and bitmap file formats. The simplest file format in the NETPBM family is the portable bitmap format, or PBM. For example, the following string is a prefix of a PBM file that represents an image that is 100 pixels wide and 60 pixels high:

```
P4 100 60
OIOOOOOOOOIIIOIIIIOOIIIIIIIIOOO...
```

A valid PBM file starts with the magic number "P4", followed by two integers n and m that specify the width and height of the

bitmap. The magic number and digits are separated by whitespace. Finally, a single newline marks the beginning of the $n \times m$ bits that constitute the black-and-white bitmap image. A zero-bit corresponds to a white pixel; a one-bit corresponds to a black pixel.

Using our combinators, it is straightforward to define the PBM file format:

```
pbm : Format
pbm = char 'P' ≫
      char '4' ≫
      char ' ' ≫
      Base NAT ≫= λn →
      char ' ' ≫
      Base NAT ≫= λm →
      char '\n' ≫
      Base (VEC (VEC BIT m) n) ≫= λbs →
      End
```

Note the dependency between the header data and the body of the bitmap: we only know how many bits to expect after having parsed the header.

This description of the PBM file format is not quite complete. We have assumed that a single space separates the magic number, width, and height. We will come back to this point, but defer any discussion for the moment.

3.4 Generic parsers

A file format is not of much use by itself. We can define a parser for any file format by induction on the *Format* data type.

```
parse : (f : Format) → List Bit → Maybe (⟦f⟧, List Bit)
parse Bad bs          = Nothing
parse End bs          = Just (unit, bs)
parse (Base u) bs     = read u bs
parse (Plus f1 f2) bs with parse f1 bs
... | Just (x, cs)    = Just (Inl x, cs)
... | Nothing with parse f2 bs
... | Just (y, ds)    = Just (Inr y, ds)
... | Nothing         = Nothing
parse (Skip f1 f2) bs with parse f1 bs
... | Nothing         = Nothing
... | Just (_, cs)    = parse f2 cs
parse (Read f1 f2) bs with parse f1 bs
... | Nothing         = Nothing
... | Just (x, cs) with parse (f2 x) cs
... | Nothing         = Nothing
... | Just (y, ds)    = Just (Pair x y, ds)
```

Most of the code should not require explanation. The *unit* constructor is the sole inhabitant of the *Unit* type. The base cases are trivial.

Agda has some syntactic sugar to avoid too much repetition when using the **with**-rule. If we do not want to repeat the entire left-hand side of a function definition, we can replace it with an ellipsis. This is particularly useful when we use non-dependent pattern matching, i.e., when we do not introduce any equalities between values.

The case for the *Plus* constructor attempts to parse the format f_1; it will only try to parse f_2 if this fails. The case for the *Skip* and *Read* constructors resemble one another: both start by parsing their first argument, but treat the result differently. Where the *Skip* constructor discards the result of parsing f_1, the *Read* passes it on to the second parser. In both cases, we continue parsing with any remaining bits. If the first parser fails, even in the *Skip* case, the input does not adhere to the specified format.

Clearly this code could profit tremendously from simple abstractions, such as the *Maybe* monad. For the sake of presentation, however, we felt that we would rather suffer this obvious repetition than introduce too many abstractions.

3.5 Generic printers

Using the same file format universe, we can also define a generic print function:

```
print : (f : Format) → ⟦f⟧ → List Bit
print Bad ()
print End _                = Nil
print (Base u) x           = toBits (show x)
print (Plus f1 f2) (Inl x) = print f1 x
print (Plus f1 f2) (Inr x) = print f2 x
print (Read f1 f2) (Pair x y) =
  append (print f1 x) (print (f2 x) y)
```

The pattern () in the first line expresses that $[\![Bad]\!]$ is not inhabited: correspondingly we do not need to write the right-hand side of the function. In the case for the *Base* constructor, we have assumed there is a function, *toBits*, to convert a string to a list of bits. The cases for the *Plus* and *Read* constructors should be fairly unremarkable.

The case for the *Skip* constructor, however, is problematic. Printing *Skip* f_1 f_2 should print a value of type $[\![f_1]\!]$ and one of type $[\![f_2]\!]$; unfortunately, the type $[\![Skip\ f_1\ f_2]\!]$ only provides us with a value of type $[\![f_2]\!]$. To successfully print both these values, we need to get our hands on a value of type $[\![f_1]\!]$.

The solution is to change the type of the *Skip* constructor as follows:

```
Skip : (f : Format) → ⟦f⟧ → Format → Format
```

The *Skip* constructor is typically used to avoid separators, magic numbers, checksums and so forth. Somehow, the parts of a file format that we skip should contain no new information. This version of the *Skip* constructor reflects this: the value of type $[\![f]\!]$ included in the file format is the value to output when you print the data structure. It does not need to be a constant, but may depend from the data that has been parsed so far, as is the case for a checksum.

The corresponding case for the *print* function now becomes straightforward to define:

```
print (Skip f1 v f2) x =
  append (print f1 v) (print f2 x)
```

To print a value x in the interpretation of a file format *Skip* f_1 v f_2, you print the default value v of type $[\![f_1]\!]$, followed by the result of printing x of type $[\![f_2]\!]$. Of course, we also need to update our *parse* function to deal with this new definition of *Skip*.

3.6 Discussion

We have deliberately chosen a fairly minimal set of constructors that illustrate the viability of our approach. Although we have not modelled all the constructs of the data description calculus (Fisher et al. 2006), it should be clear how to extend the code we have presented here to deal with most of the constructors we have omitted.

Most notably, our *Format* data type does not have any form of recursion. We could define a *many* combinator as follows:

```
many : Format → Format
many f = Plus (Read f (λ_ → many f)) End
```

This definition, however, uses general recursion. The Agda compiler warns us that it may fail to terminate; the Agda type checker may diverge when trying to type check file formats that use *many*.

For example, evaluating $[\![many\ (Base\ CHAR)]\!]$ corresponds to constructing the following infinite type:

$$\begin{array}{l} Either \\ \quad (Sigma\ Char\ (\lambda x \rightarrow \\ \quad\quad (Either\ (Sigma\ Char\ (\lambda y \rightarrow ...))) \\ \quad\quad\quad Unit) \\ \quad Unit \end{array}$$

As we saw in our specification of the PBM file format, however, we really want to parse a sequence of whitespace characters.

The simplest solution is to extend the *Format* data type with a new constructor $Many : Format \rightarrow Format$ that greedily parses as many subsequent occurrences of its argument as possible.

The more general solution, however, would be to extend our universe with variables and a least-fixed point operation. This would enable us to describe not only lists, but a much wider class of data types. We have refrained from doing so as the resulting universe must deal with variable binding. Although the solution is not terribly complex (Morris et al. 2004), we felt the technical overhead would distract from the bigger picture.

Readers familiar with generic programming may not be surprised by our results. Systems such as Generic Haskell (Hinze and Jeuring 2003) have already shown how to write generic read and show functions for a universe closed under sums and products. In contrast to Generic Haskell, however, our universe is closed under *dependent* pairs, not simple products. This dependency is very important when parsing binary data, as our pbm example illustrates. Furthermore, we show how dependently-typed programming languages support generic programming without resorting to preprocessors, in contrast to Generic Haskell.

These parsers are, of course, closely related to monadic parser combinator libraries such as Parsec (Leijen and Meijer 2001). In Parsec, for example, you can also parse a number n, followed by n bits:

$$\begin{array}{l} parseVec = \mathbf{do}\ n \leftarrow parseInt \\ \qquad\qquad\quad\ \ xs \leftarrow count\ n\ parseBit \\ \qquad\qquad\quad\ \ return\ (n, xs) \end{array}$$

This parser will return a value of type $(Int, [Bit])$; the relationship between the second and first element of the pair is irretrievably lost. Our parsers, on the other hand, return a dependent pair that preserves this information.

The data type $[\![f]\!]$ associated with a format f will always be a nested tuple of values. Manipulating such data types may become rather tiresome. Fortunately, as the previous section illustrates, it is straightforward to define a view on the generated data. For example, we may want to view $[\![pbm]\!]$ as a record with meaningful labels, $width$, $height$, and $bitmap$, and associated projection functions.

Many data description languages support various error reporting and error recovery features. PADS (Fisher and Gruber 2005), for example, has two different functions of type $Format \rightarrow Set$: one corresponds to our $[\![\_]\!]$ function; the other decorates the resulting data type with diagnostic information about errors encountered during parsing. Defining this second interpretation of file formats, and updating our parsers accordingly would improve the error messages significantly. We can, of course, also use existing techniques to improve our $parse$ function's error messages (Swierstra and Duponcheel 1996).

We would like to emphasise that this domain-specific language enforces several important properties. The type of the $parse$ function, for example, ensures that the parser for a file format f will return a value of type $[\![f]\!]$ upon success. In the existing work on the data description calculus (Fisher et al. 2006), this is an important meta-theoretical result that requires some effort to prove. We know this important property holds by construction: *our types give us meta-theory for free*!

This section shows how important it is to compute *types* from *data* when writing generic programs. As we shall see, however, the same problem also appears in a much more mundane setting.

4. Relational algebra

Databases are everywhere. When you book a flight, order a book, or rent a movie online, all you are really doing under the hood is querying and updating a database.

For this reason, a programming language must be able to interface with a database. Most of the time such an interface consists of a pair of functions to send a *request*—as a simple string containing an SQL query—and to receive a *response*—usually in the form of a string or some dynamic type. While this approach is simple to implement, it has numerous drawbacks:

- This interface is unsafe: there are no static checks on the queries. It is all too easy to formulate a syntactically incorrect or semantically incoherent query; an unexpected response from the database server results in a runtime error.
- Programmers need to learn another language. Moreover, they need to switch from one language to another in the body of a same function.

To address these issues, there have been several proposals to embed a domain-specific language for database queries in Haskell (Leijen and Meijer 1999; Bringert et al. 2004). Each of these proposals provides a set of combinators to construct queries. These queries in Haskell can be 'compiled' to a string, corresponding to an SQL query that can be sent to a database server.

Unfortunately, all the typed database bindings to Haskell have one or more of the following drawbacks:

- They struggle to express all the concepts of relational algebra. For example, the *join* and *cartesian product* of two tables is notoriously hard to type.
- Embedding the type system of the query language in Haskell is not easy. Existing bindings must either rely on unsupported features, such as extensible records, or type-level programs written using multi-parameter type classes with functional dependencies.
- Safe bindings rely on static knowledge of the database a program will query. This usually manifests itself in form of a preprocessor that connects to the database and generates the Haskell type declarations that represent the values stored in the database.

As a result, there is no widely adopted set of typed database bindings for Haskell. Many popular bindings, such as HDBC and Takusen, resort to some form of dynamic typing by using a single Haskell data type to represent all SQL's types. Any type errors a programmer makes, will only be detected dynamically.

All these limitations share the same origin: to embed a domain-specific language, you need to embed a domain-specific type system in the type system of your host language. Haskell's type system, however, is fundamentally different from that of a database query language. Shoehorning a query language's type system into Haskell requires significant amounts of type hackery.

In this section, we sketch how to write a domain-specific embedded language for relational algebra in Agda. In contrast to all the existing Haskell implementations, our combinator library is both *safe* and *totally embedded*, that is, it does not rely on any form of preprocessor. The resulting code provides similar guarantees to the

type-safe Haskell bindings, yet the code significantly shorter and easier to understand.

4.1 Schemas, Tables, and Rows

A relational database consists of a collection of *tables*. For example, the makers of the British television program *TopGear* time how long it takes them to drive various cars around their test track. A database storing their results could contain the following table:

Model	Time	Wet
Ascari A10	1:17.3	False
Koenigsegg CCX	1:17.6	True
Pagani Zonda C12 F	1:18.4	False
Maserati MC12	1:18.9	False

This table stores the best lap time of several different kinds of car. Besides storing the lap time itself, it also records if the track was wet when the time was recorded, as this may influence the lap time. As this example illustrates, a table may contain different types of information: a string corresponding to the car's make and model; a time written in minutes, seconds, and deciseconds; the track conditions is represented by a boolean.

A *schema* describes the type of a table. It consists of a set of pairs of column names and types:

$$\begin{aligned}&\mathit{Attribute} : \mathit{Set}\\&\mathit{Attribute} = (\mathit{String}, U)\\&\mathit{Schema} : \mathit{Set}\\&\mathit{Schema} = \mathit{List}\ \mathit{Attribute}\end{aligned}$$

We do not allow *any* type to occur in a Schema, but restrict ourself to the universe (U, el) from the previous section. Most database servers only support a small number of types, such as booleans, integers, times, dates, and (fixed-width) strings. It should be clear how to define a universe to capture the types supported by any particular database server.

One choice of schema for our example table would be:

$$\begin{aligned}&\mathit{Cars} : \mathit{Schema}\\&\mathit{Cars} = \mathit{Cons}\ (\texttt{"Model"}, \mathit{VEC}\ \mathit{CHAR}\ 20)\\&\qquad(\mathit{Cons}\ (\texttt{"Time"}, \mathit{VEC}\ \mathit{CHAR}\ 6)\\&\qquad(\mathit{Cons}\ (\texttt{"Wet"}, \mathit{BOOL})\ \mathit{Nil}))\end{aligned}$$

Here we have chosen the car's make and time as fixed-width strings of a certain length. This is, of course, a rather questionable choice of schema: would it not be better to use a triple of integers, for example, to represent the time? We will come back to this point at the end of this section.

We can now define a table to consist of a list of *rows*. A row is a sequence of values, in accordance with the types dictated by the table's schema. The $\mathit{EmptyRow}$ constructor is corresponds to a row with an empty schema; to create a row in a schema of the form $\mathit{Cons}\ (\mathit{name}, u)\ s)$, you need to provide an element of type $\mathit{el}\ u$, together with a row adhering to the schema s.

$$\begin{aligned}&\mathbf{data}\ \mathit{Row} : \mathit{Schema} \to \mathit{Set}\ \mathbf{where}\\&\quad\mathit{EmptyRow} : \mathit{Row}\ \mathit{Nil}\\&\quad\mathit{ConsRow} : \mathbf{forall}\ \{\mathit{name}\ u\ s\} \to\\&\qquad\mathit{el}\ u \to \mathit{Row}\ s \to \mathit{Row}\ (\mathit{Cons}\ (\mathit{name}, u)\ s)\\&\mathit{Table} : \mathit{Schema} \to \mathit{Set}\\&\mathit{Table}\ s = \mathit{List}\ (\mathit{Row}\ s)\end{aligned}$$

For example, the third row in the table above could be written:

$$\begin{aligned}&\mathit{zonda} : \mathit{Row}\ \mathit{Cars}\\&\mathit{zonda} = \mathit{ConsRow}\ \texttt{"Pagani Zonda C12 F "}\\&\qquad(\mathit{ConsRow}\ \texttt{"1:18.4"}\\&\qquad(\mathit{ConsRow}\ \mathit{False}\ \mathit{EmptyRow}))\end{aligned}$$

Here we have taken the syntactic liberty of writing string literals instead of vectors containing characters—this is not valid Agda as it stands.

Dealing with such heterogeneous lists is the first stumbling block for many Haskell database bindings. The corresponding Haskell code is much more difficult to comprehend than this simple definition (Kiselyov et al. 2004).

4.2 Setting up a database connection

Before you can actually query the database, you will need to set up a connection with a database server. Many database interfaces for Haskell provide a function of the following type:

$$\mathit{connect} : \mathit{ServerName} \to \mathit{IO}\ \mathit{Connection}$$

Here $\mathit{ServerName}$ is simply a type alias for String. The constructors of the $\mathit{Connection}$ type are not visible to the library's users. Instead, it can be used to send a string corresponding to some SQL query to a particular database:

$$\mathit{query} : \mathit{String} \to \mathit{Connection} \to \mathit{IO}\ \mathit{String}$$

Once you have set up a connection, however, your types carry no information whatsoever about the database to which you are connected. As a result, you have no static checks on how you choose to interpret a database's response to your query.

Using dependent types, we can be much more precise about the data stored in a table. A much better choice for the connect function that is exposed to the library's users is:

$$\begin{aligned}&\mathit{Handle}\ : \mathit{Schema} \to \mathit{Set}\\&\mathit{connect} : \mathit{ServerName} \to \mathit{TableName} \to\\&\qquad(s : \mathit{Schema}) \to \mathit{IO}\ (\mathit{Handle}\ s)\end{aligned}$$

Instead of returning a connection to some unknown database, the user sets up a connection to a particular table of the database. He also states the expected Schema of this particular table. The $\mathit{connect}$ function then returns a Handle to that particular table, with a type ensuring that this table respects the schema s. In practice, a user will always want to set up a connection to several tables in one step. We have chosen this simplification for the sake of presentation: it is by no means a limitation of our approach.

This connect function does more work than its simply-typed counterpart we mentioned previously. In addition to setting up the connection with the database server, it asks for a description of its table argument. Database servers, when prompted to describe a table, respond with a string such as:

```
Name        Type
----------- ----------
MODEL       CHAR(20)
TIME        CHAR(6)
WET         BOOL
```

It should be clear how to parse this response and build a value of type Schema. This value is then compared to the Schema provided by the user. If the two schemas are the same, a Handle to the table is returned. If the two schemas are different, $\mathit{connect}$ will result in a *runtime exception* in the IO monad. This exception occurs when the database the programmer attempted to connect to does not have the schema he expected. This kind of exception is inherent to working with IO: the real world can always behave unexpectedly.

However, there are two key guarantees that the type system provides:

- The only point where this Schema mismatch can occur is during a call to $\mathit{connect}$. Any subsequent *queries* using the Handle are safe.
- In particular, if the Schema passed to the $\mathit{connect}$ function is the same as the Schema of the table we connect to, the

program *cannot go wrong*, provided the data base schema does not change in the meantime, the connection is not lost, etc.

4.3 Constructing queries

Once we have set up a connection, we want to query the database. Rather than model any particular flavour of SQL, we will show how to embed *relational algebra operators* in Agda. For the sake of simplicity, we have chosen only to model five operations: selection, projection, cartesian product, set union, and set difference.

To introduce these operations, we define the following type RA. An expression of type $RA\ s$ corresponds to a query that will return a table with schema s—that is, we know statically exactly what kind of table to expect when we execute any given query.

```
data RA : Schema → Set where
  Read    : forall {s} → Handle s → RA s
  Union   : forall {s} → RA s → RA s → RA s
  Diff    : forall {s} → RA s → RA s → RA s
  Product : forall {s s'} → {So (disjoint s s')}
    → RA s → RA s' → RA (append s s')
  Project : forall {s} → (s' : Schema)
    → {So (sub s' s)} → RA s → RA s'
  Select  : forall {s} → Expr s BOOL
    → RA s → RA s
```

Besides the five primitive operations we mentioned above, we have one constructor, $Read$, that simply reads the table associated with a $Handle$. The $Union$ and $Diff$ constructors correspond to the set union and set difference of two tables. The $Product$, $Project$, and $Select$ constructors are more interesting.

The $Product$ constructor takes the *cartesian product* of two tables. For any two tables $t1$ and $t2$, the cartesian product of $t1$ and $t2$ can be specified as follows in Haskell:

```
do r1 ← t1
   r2 ← t2
   return (appendRow r1 r2)
```

Here we use the list monad to select any pair of rows from $t1$ and $t2$; the $appendRow$ function appends two rows in the obvious manner. In order to take the cartesian product of two tables, however, their schemas must be disjoint. It is easy to write a function that checks if two schemas are disjoint:

```
disjoint : Schema → Schema → Bool
```

But how should we enforce that $disjoint\ s\ s'$ must be $True$? The solution is to require a proof of $So\ (disjoint\ s\ s')$, where So is defined as follows:

```
So : Bool → Set
So True  = Unit
So False = Empty
```

If p is true, the proof $So\ p$ is trivial; if not, there is no way to pass an argument of the right type.

Note that the proof argument of type $So\ (disjoint\ s\ s')$ to the $Product$ constructor is implicit. Agda is clever enough to automatically fill in implicit arguments of type $Unit$. If the schemas are known at compile time, $disjoint\ s\ s'$ will compute to either $True$ or $False$. Consequently, $So\ (disjoint\ s\ s')$ will evaluate to $Unit$ or $Empty$. As a result, a programmer will *never* have to worry about such proofs for closed schemas. When no such proof exists, Agda will complain that it cannot fill in implicit arguments—the programmer is then responsible for fixing this.

The $Project$ constructor projects out some sub-schema of a given table. For example, if we want to know the model of all the cars TopGear has tested, we can formulate this query as follows:

```
Models : Schema
Models = Cons ("Model", VEC CHAR 20) Nil

models : Handle Cars → RA Models
models h = Project Models (Read h)
```

Just as the $Product$ constructor required its two schema arguments to be disjoint, the $Project$ constructor requires its second schema argument to be a subset of its first schema argument. For the sake of simplicity, we use a value s' of type $Schema$ to describe the projected fields, but we could replace it by a list of the names of the fields. The type of these field could then be recovered from the schema from which we project.

Finally, the $Select$ constructor filters the result of a query. Leijen and Meijer (1999) have already shown how to use phantom types to safely embed the operators of SQL in Haskell:

```
data Expr : Schema → U → Set where
  equal    : forall {u s} → Expr s u → Expr s u
    → Expr s BOOL
  lessThan : forall {u s} → Expr s u → Expr s u
    → Expr s BOOL
  _!_ : (s : Schema) → (nm : String)
    → {So (occurs nm s)}
    → Expr s (lookup nm s p)
  ...
```

We follow their lead and index the data type for SQL expressions, $Expr$, by their return type, represented by a value of type U. We also index these expressions by a $Schema$ describing the *attributes* to which the expression may refer. SQL supports a small number of primitive operations for comparing values for equality, boolean conjunction, and so forth.

The only particularly interesting operation, $\_\,!\,\_$, looks up an attribute in the schema. Once again, we require an implicit proof that the name of the attribute does indeed occur in the schema. As we saw previously, this proof is automatically discharged by Agda if the s and nm are known at compile time. Using the $\_\,!\,\_$ constructor returns a value of type $Expr\ s\ (lookup\ nm\ s\ p)$, where the $lookup$ function finds the U associated with the name nm. In order for this function to be total we need the implicit proof p. Without p, we do not know whether or not the name occurs in the schema.

For instance, we may want to query the database for all the models of cars that have been tested under wet conditions:

```
wet : Handle Cars → RA Models
wet h = Project Models (Select (Cars ! "Wet") (Read h))
```

This expression does not require any proofs. Even though the constructors $Project$ and $Select$ have implicit proof arguments, we *compute* the proof using the sub and $occurs$ functions; the proof itself turns out to be so trivial that Agda can be fill it in automatically. This automation of trivial proofs is a key point in the design of user-friendly embedded type systems.

As we mentioned previously, we have taken a very minimal set of relational algebra operators. It should be fairly straightforward to add operators for the many other operators in relational algebra, such as the natural join, θ-join, equijoin, renaming, or division, using the same techniques. Alternatively, you can define many of these operations in terms of the operations we have implemented in the RA data type.

4.4 Executing queries

Once we have formulated a query, we would like to pass it to the database server. We could define a function that generates the SQL query associated with every relational algebra expression:

```
toSQL : forall {s} → RA s → String
```

We can pass this $String$ to the database server and wait for the result. However, this function throws away precious type information! We can do much better. The function our library should export to execute a query should really have the following type:

$$query : \{ s : Schema \} \to RA\ s \to IO\ (List\ (Row\ s))$$

The $query$ function uses $toSQL$ to produce a query, and passes this to the database server. When the server replies, however, we know exactly how to parse the response: we know the schema of the table resulting from our query, and can use this to parse the database server's response in a type-safe manner. The type checker can then statically check that the program uses the returned list in a way consistent with its type.

4.5 Discussion

There are many, many aspects of this proposal that can be improved. Some attributes of a schema contain $NULL$-values; we should close our universe under $Maybe$ accordingly. Some database servers silently truncate strings longer than 255 characters. We would do well to ensure statically that this never happens. Our goal, however, was not to provide a complete model of all of SQL's quirks and idiosyncrasies: we want to show how a language with dependent types can shine where Haskell struggles.

Our choice of $Schema$ data type suffers from the usual disadvantages of using a list to represent a set: our $Schema$ data type may contain duplicates and the order of the elements matters. The first problem is easy to solve. Using an implicit proof argument in the $Cons$ case, we can define a data type for lists that do not contain duplicates. The type of $Cons$ then becomes:

$$\begin{aligned} Cons : (nm : String) &\to (u : U) \to (s : Schema) \\ &\to \{ So\ (\neg\ (elem\ nm\ s)) \} \\ &\to Schema \end{aligned}$$

The second point is a bit trickier. The real solution would involve quotient types to make the order of the elements unobservable. As Agda does not support quotient types, however, the best we can do is parameterise our constructors by an additional proof argument, when necessary. For example, the $Union$ constructor could be defined as follows:

$$\begin{aligned} Union : \mathbf{forall}\ \{ s\ s' \} &\to \{ So\ (permute\ s\ s') \} \\ &\to RA\ s \to RA\ s' \to RA\ s \end{aligned}$$

Instead of requiring that both arguments of $Union$ are indexed by the same schema, we should only require that the two schemas are equal up to a permutation of the elements. Alternatively, we could represent the $Schema$ using a data structure that fixes the order in which its constituent elements occur, such as a trie or sorted list.

Finally, we would like to return to our example table. We chose to model the lap time as a fixed-length string—clearly, a triple of integers would be a better representation. Unfortunately, most database servers only support a handful of built-in types, such as strings, numbers, bits. There is no way to extend these primitive types. This problem is sometimes referred to as the *object-relational impedance mismatch*. We believe the generic programming techniques and views from the previous sections can be used to marshall data between a low-level representation in the database and the high-level representation in our programming language.

5. Conclusions and Related work

Related work There are several other type systems that enrich simply typed languages with more indexing information (Freeman and Pfenning 1991; Sheard 2005; Peyton Jones et al. 2006). Using GADTs in Haskell, for example, programmers can write a data type of vectors. However, many of the examples from this paper revolve around computing new *types* from *data*. In a dependently-typed language we can, for example, download a file format description from the web and compute the type of its associated parser; or we can connect to a data base and compute the type of its tables. The static index information other systems provide cannot easily cope such examples that freely mix types, values, and computation.

Programming with dependent types is subject to active research. Although we have chosen to use Agda throughout this paper, there are many alternatives, such as Epigram or Coq, especially augmented with the Program tactic (Sozeau 2007). The programs in this paper could have been written in any of these systems; each system has its own particular strengths and weaknesses.

There are several other papers about programming with dependent types. The Epigram lecture notes by McBride (2004) are essential reading. There are a few other studies of domain-specific languages in the context of dependent types, including stack machines (McKinna and Wright), locality-aware multi-core programs (Swierstra and Altenkirch 2008), and interpreters (Augustsson and Carlsson).

Conclusions What is the Power of Pi? We believe that many of the advantages of programming with dependent types are covered by the following three points:

Precise data types A good program never crashes. Programmers should be as accurate as possible when defining data types and avoid writing partial functions. Haskell data types can contain junk values: what is the head of an empty list? Precise dependent data types liberate programmers from worrying what to do when 'the impossible occurs.'

Views Such precise data types are of little value if working with them becomes tedious. Most functional languages allow the designers of domain-specific embedded languages to carefully engineer the *constructors* of their domain. Various combinators and functions can then help build larger programs. There is, however, no uniform way to abstract over patterns that *destruct* domain-specific data. As Epigram demonstrates, languages with dependent types can be used to define compositional domain-specific views.

Universes Statically typed functional languages have started to exploit types when writing programs. This is illustrated by advances in generic programming or the many applications of Haskell's type classes. Many of the examples in this paper revolve around universe constructions that enable us to *program* with *types*.

It has been more than ten years since the first work about domain-specific embedded languages (Elliott and Hudak 1997) sparked off a new field of research. Functional programming found new applications in randomised testing (Claessen and Hughes 2000), financial markets (Peyton Jones et al. 2000), server side webscripting (Thiemann 2004), hardware design (Bjesse et al. 1998), and reactive programming (Nilsson et al. 2002). More recently, this avenue of research seems to have dried up. In its place, there are more and more papers about type systems designed to solve a very specific problem.

This paper proposes to drastically break with this trend. The techniques we have presented enable us to embed such domain-specific type systems in a dependently-typed host language. This does not require writing a new compiler and we inherit the metatheory of our host language for free. Furthermore, this provides us with a single semantic framework for exploring the design space. Most importantly, however, it revives what functional programming is really about: writing programs, not designing typing rules. For that reason, if nothing else, dependent types matter.

Acknowledgments

We would like to express our sincerest gratitude to Thorsten Altenkirch for his encouragement; Lennart Augustsson for his comments about relational algebra; James Chapman for our discussions about file formats and Erlang; Peter Hancock for teaching us everything we know about universes; Ulf Norell for his fantastic new incarnation of Agda and his tireless tech-support; Conor McBride for his inspirational tutelage; and last but not least, we would like to thank Björn Bringert, Nils Anders Danielsson, Isaac Dupree, Chris Eidhof, Andy Gill, Michael Greenberg, Eelco Lempsink, Mads Lindstrøm, Andres Löh, Matthew Naylor, Henrik Nilsson, Bernie Pope, Matthieu Sozeau, Stephanie Weirich, Brent Yorgey, and our anonymous reviewers for their invaluable feedback.

References

Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: a well-typed interpreter. Unpublished manuscript.

Godmar Back. Datascript - a specification and scripting language for binary data. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, 2002.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, 1998.

Björn Bringert, Anders Höckersten, Conny Andersson, Martin Andersson, Mary Bergman, Victor Blomqvist, and Torbjörn Martin. Student paper: HaskellDB improved. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, 2004.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP '97: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, 1997.

Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. *SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, 1991.

Galois, Inc. *Cryptol Reference Manual*, 2002.

Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 1998.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, 2004.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, October 1999.

Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht, 2001.

Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *LNCS-Tutorial*, pages 130–170. Springer-Verlag, 2004.

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

Peter J. McCann and Satish Chandra. Packet types: abstract specification of network protocol messages. In *SIGCOMM '00: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2000.

James McKinna and Joel Wright. A type-correct, stack-safe, provably correct expression compiler in Epigram. Accepted for publication in the Journal of Functional Programming.

Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *Types for Proofs and Programs*, volume 3839 of *LNCS*. Springer-Verlag, 2004.

Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, 2002.

Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering. In *ICFP '00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, 2006.

Tim Sheard. Putting Curry-Howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, 2005.

Matthieu Sozeau. Subset coercions in Coq. In *Types for Proofs and Programs*, volume 4502 of *LNCS*. Springer-Verlag, 2007.

S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, 1996.

Wouter Swierstra and Thorsten Altenkirch. Dependent types for distributed arrays. In *Proceedings of the Ninth Symposium on Trends in Functional Programming*, 2008.

The Agda Team. Agda homepage. `http://www.cs.chalmers.se/~ulfn/Agda`, 2008.

Peter Thiemann. Server-side web programming in WASH. In *Advanced Functional Programming*, volume 3622 of *LNCS-Tutorial*. Springer-Verlag, 2004.

Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *14th Symposium on Principles of Programming Languages*, 1987.