Dependent Type Providers

David Raymond Christiansen
IT University of Copenhagen
drc@itu.dk

Abstract
Type providers [16], pioneered in the F# programming language, are a practical and powerful means of gaining the benefits of a modern static type system when working with data schemas that are defined outside of the programming language, such as relational databases. F# type providers are implemented using a form of compile-time code generation, which requires the compiler to expose an internal API and can undermine type safety. We show that with dependent types it is possible to define a type provider mechanism that does not rely on code generation. Using this mechanism, a type provider becomes a kind of generic program that is instantiated for a particular external schema, which can be represented using an ordinary datatype. Because these dependent type providers use the ordinary abstraction mechanisms of the type system, they preserve its safety properties. We evaluate the practicality of this technique and explore future extensions.

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

Keywords Dependent Types; Type Providers; Metaprogramming; Generic Programming

1. Introduction
Version 3.0 of the F# programming language has a new feature called Type Providers, which is a mechanism for representing the schemas of external data sources in the type system. A type provider is a compiler plugin that performs compile-time code generation, producing both .NET types and program code that uses these types, in response to external information. While F# type providers are very useful, they suffer from two key disadvantages: (1) the generated code may not be well-typed and generated types are not necessarily well-formed; and (2) the code generation process goes outside the ordinary semantic world of the language, giving a view into F# implementation internals.

In a language with dependent types, it is possible to implement a mechanism for type providers in a manner that requires no code generation and thus does not suffer from the two mentioned disadvantages. This mechanism exploits the technique of generic programming with universes [1, 3].

We have implemented dependent type providers in the programming language Idris [5, 7]. Idris is a dependently typed, strict-by-default language with strong syntactic similarities to Haskell. While this paper uses Idris’s syntax and discusses dependent type providers in that context, the presented technique is applicable to any dependently-typed functional programming language that has a facility for effectful computation.

While most work on generic programming focuses on techniques for defining generic programs, dependent type providers focus on having the compiler instantiate a generic program (the library) and then ensure that the user’s program is consistent with said instantiation.

Our contributions are:
- We describe dependent type providers in Section 3;
- We analyze their advantages and disadvantages relative to F#'s type providers in Section 4;
- We describe their implementation in Idris, as a straightforward extension to Idris’s elaboration-based semantics in Section 5; and
- We describe and motivate future extensions to dependent type providers in Section 10.

Section 2 presents necessary background information. Additionally, we present two example type providers in Sections 6 and 7 and discuss alternative formulations of dependent type providers in Section 8. Finally, Section 9 wraps up the discussion and Section 11 discusses related work.

2. Background
2.1 F# Type Providers
Software development does not occur in a vacuum. Many programs need to make use of data that the programmer does not control, such as external databases. This external data will often be structured, with a specific set of valid field names and specific patterns of use. However, that structure is not immediately available to programming languages’ type systems. When accessing external data, developers are stuck with an essentially dynamically typed paradigm. The type system is unable to assist them in accessing their data correctly.

It is certainly possible to manually write statically-typed bindings to the external data source that reflect the schema. However, the maintenance burden of these bindings in the face of rapidly changing external schemas is high. While traditional code generation can be of assistance in dealing with schema changes, it has a number of drawbacks. First, it is cumbersome: an extra tool must be run during the build process. Second, the code generator has no way to know which parts of the external data will be accessed, so it must generate types for each entity in the external data. In the case of very large schemas, this can be a significant practical problem.

For example, the publicly-curated database Freebase [4] has 24,216 types at the time of writing, ranging from scientific informa-
Idris supports defining datatypes using both the traditional Haskell syntax and a GADT-style syntax. In the interest of clarity and space, we use the former when possible.

The type of types is called Type, rather than *. Each instance of Type has an implicit level, inferred by the compiler. Levels are cumulative — everything in Type, is also in Type\_e\_1.

In Idris, free variables that start with a lower-case letter are converted to implicit arguments. It is also possible to mark an argument as implicit by surrounding it in curly braces. Additionally, ad-hoc overloading of function and constructor names is allowed, provided that they occur in separate namespaces, and types are used to disambiguate them automatically. The constructor names nil and (:::) are special: any datatype defined with constructors that use them is able to be constructed with list literal syntax.

### 2.3 Generic Programming with Universes

The technique of generic programming with universes is well-known in the dependently-typed programming community. Here, we provide a brief introduction. More detailed explanations can be found in Altenkirch and McBride [1], and Andjelkovic’s M.Sc. thesis [2] contains an especially accessible introduction.

A universe consists of a datatype whose elements describe a particular collection of types, and an interpretation function that maps elements of the datatype, called codes, to actual types. Generic programs are defined by recursion on the codes.

For example, a simple universe \( U \) consisting of integers, lists, and vectors is defined in Idris as follows:

```idris
data U = INT | LIST U | VECT U Nat

interp : U -> Type
interp INT = Int
interp (LIST t) = List (interp t)
interp (VECT t n) = Vect (interp t) n
```

We follow the convention of using capital letters to represent codes in a universe, identifiers beginning with capital letters to represent ordinary datatypes and constructors, and lowercase identifiers to represent functions, type variables, and other identifiers. The type \( \text{Vec} \ a \ n \) is a kind of list, familiar to anyone who has read an introductory tutorial on dependent types, that is guaranteed to be precisely \( n \) elements long.

We can write a generic function by pattern-matching on an argument of type \( U \). Dependent pattern matching can use information gained through a case split to induce equalities in the types of the other arguments. In particular, the call to \( \text{interp} \) will be reduced, yielding the actual type of the next argument. A generic equality function defined over all of the types in \( U \) follows:

```idris
equal : (t : U) -> interp t -> interp t -> Bool
equal INT i1 i2 = i1 == i2
equal (LIST t) [] [] = True
equal (LIST t) (x :: xs) (y :: ys) =
  equal t x y &&
  equal (LIST t) xs ys

equal (VECT t n) [] [] = False
equal (VECT t n) (x :: xs) (y :: ys) =
  equal t x y &&
  equal (VECT t n) xs ys
```

This simple universe does not allow implementations of equal to take advantage of the shared structure of \( \text{List} \) and \( \text{Vect} \). This code duplication can be removed using a technique similar to generic programming \( \text{à la} \) Hinze [11]. This is achieved by using a universe with codes representing sums, products, parameters, recursive binding, and so forth. While the universe technique can be-

---

1 The number of types can be seen at [http://www.freebase.com/query using the query [{"id": null, "name": null, "type": "/type/type", "return": "count"}]
come quite involved when representing dependent types and other rich types systems, we expect that most type providers will be used to interface with systems that are relatively inexpressive.

3. Dependend Type Providers

Because the .NET type system lacks dependent types, code generation is a necessary part of F# type providers. However, languages with dependent types can do better. With dependent types, one can simply return types from functions. Also, with inductive families[10], it is possible to use datatype indices to exert a large degree of control over how the types behave.

We have implemented dependent type providers in the Idris language. However, this section describes a mechanism for type providers that, in principle, should be realizable in any dependently-typed functional language that has some notion of effectful programs.

A dependent type provider consists of a universe of datatypes, a generic library defined on that universe, and an effectful program that constructs codes in the universe. However, in simple cases such as the examples in this chapter, a dependent type provider may just be an effectful program that returns a type directly. During type checking, the effectful program is run, and its result is extracted to a top-level definition. The rest of the program can then use the resulting term during type checking. Each time the program is type checked, the term will be evaluated again, to ensure that changes to external data sources are reflected. If necessary for efficiency or usability, type providers can implement caching mechanisms.

As implemented in Idris, a provider of some type t is an expression of type IO (Provider t). IO is necessary because the type provider may have side effects. The type Provider t is isomorphic to Either String t:

data Provider t = Error String | Provide t

We have added the following syntax for invoking type providers to Idris:

%provide (x : T) with p

The name x will be bound to the result of the type provider, and it will have type T, which can be either be Type or some specific type. The actual type provider is the expression p. When the type checker encounters this declaration, the provider expression p is checked to have type IO (Provider T). Next, the expression p is executed and the result of the execution is extracted from the IO monad. If the result is Provide y, then the declaration x : T and the definition x = y are added to the global context.

For example, the following code snippet defines and uses a type provider that depends on the content of a particular file. If the file contains the string "Integer", then the provider returns the type Int. Otherwise, it returns the type Nat. The definition takes advantage of Idris’s overloading of integer literals to compile in either case.

  strToType : String -> Type
  strToType "Integer" = Int
  strToType _ = Nat

fromFile : String -> IO (Provider Type)
fromFile fname =
  do str <- readFile fname
     return (Provide (strToType str))

%provide (T1 : Type) with fromFile "theType"

two : T1
two = 2

There are many ways in which this process can go wrong aside from obvious situations like mismatching types. First, because IO allows arbitrary effects, there is no guarantee that the provider will even terminate. In the present implementation, this will cause the type checker to fail to terminate, but the situation could be ameliorated somewhat by introducing a timeout. Likewise, if the provider fails in some other manner (for example, if it reads from a closed file), the type checker itself will fail.

In addition to exceptional failures such as a failure to terminate or a crash, type providers can signal errors with informative error messages. They do this by returning the Error constructor with an appropriate message.

An example of the full generality of this style of type provider is a type provider that causes typechecking to fail if a user claims to be under 18 years of age. This example demonstrates the unrestricted nature of the I/O actions that can be used with type providers. The demonstrated type provider, which is named adultsOnly, asks the user for his or her age. It then attempts to parse the result as an integer. If it fails, it asks repeatedly until it succeeds. When an integer is successfully parsed, adultsOnly then checks whether it is at least 18. If this is the case, then adultsOnly provides the Boolean value True, and if the user is not yet 18, then typechecking fails. Obviously, this is easy for underage programmers to defeat — nevertheless, it is an effective demonstration of the error mechanism.

  confirmAge : IO Bool
  confirmAge = do putStrLn "How old are you?"
   input <- getLine
   let age = parseInt (trim input)
   case age of
     Nothing =>
       do putStrLn "Didn’t understand"
          confirmAge
     Just x =>
       return (if x >= 18 then True else False)

adultsOnly : IO (Provider Bool)
adultsOnly =
  do oldEnough <- confirmAge
   if oldEnough
      then do putStrLn "ok"
             return (Provide True)
      else return (Error "Only adults may " ++ "compile this program")

%provide (ok : Bool) with adultsOnly

Note that this mechanism does not permit type providers to introduce new names into the global context. The name x is specified by the client of the type provider. Because the name that the provided term will be bound to and the type that it must inhabit are both explicit, implementors of type providers are not burdened by handling the complexities of names and binding. Additionally, the types assigned to top-level names are obvious when reading code that uses type providers.

Idris, similarly to Epigram[13] and Agda[17], restricts full dependent pattern matching to the left hand side of top-level definitions. Because execution can only produce ordinary terms, it can at best provide the right-hand side of a definition. Likewise, type providers cannot define new inductive data types. Provided types and terms must be constructed from already-defined constructors, type constructors, and pattern-matching definitions. This is a result of the straightforward semantics of the %provide construct: only
ordinary execution is required, and there is no manual manipulation of abstract syntax trees.

4. Safety and Expressiveness

F# type providers have essentially unlimited freedom to define new types as well as to generate code. These new types may fail to respect the ordinary rules of the F# type system — for example, they can have circular inheritance or refer to nonexistent types. Additionally, the generated code is also exempted from typechecking. That is, terms that are not well-typed can be generated by a type provider, leading to potential run-time crashes. Therefore, F# type providers are strictly less safe than classically generated code.

From a fundamentalist point of view, this should not be of concern. Accepting that types could vary based on arbitrary I/O computations entails giving up not only properties such as decidability of typechecking, but even basic assumptions such as determinism of typechecking. A type provider similar to the above adultOnly could refuse to typecheck in the afternoon, for example. Even worse, changes to the schemas of external data sources can cause a well-typed program to begin going wrong some time after compilation. As no guarantees remain, new ways to introduce errors will not be of concern to someone who has a purely theoretical approach to programming languages.

However strong the theoretical argument, this approach is not particularly well-grounded in the experience of real users of real systems. As Syme et al. [16] point out, certain unsound extensions can vastly ease the development of real software. In some sense, users of F# type providers give up a certain amount of safety in exchange for expressiveness.

The expressiveness vs. safety trade-offs of dependent type providers are different. In contrast to F# type providers, dependent type providers cannot generate new datatype definitions. They are also unable to generate new top-level pattern-matching definitions. This reduction in expressiveness is matched by an increase in safety relative to F#. Our dependent type providers can only produce terms, which may or may not be actual types. These terms are subject to the ordinary typechecking process, so users are just as safe with an Idris type provider as they would be with a build process that generates code. Concerns about determinism and decidability of typechecking are still present, however.

Because arbitrary terms can result from an Idris type provider, the expressiveness of dependent type providers is exactly the expressiveness of indexed datatypes in Idris. In other words, to the extent that a universe can represent the type in question, dependent type providers can also represent it.

A key aspect of F#’s type providers’ expressiveness is their support for what Syme et al. [16] call “design-time” aspects. By this, they mean that type providers must be supported in modern integrated development environments. Indeed, the ability to create intelligent development environments is a key argument for the industrial utility of statically-typed programming languages. This aspect of expressiveness is not yet ready to be evaluated in Idris. Currently, the tools available for writing Idris code are not anywhere near as sophisticated as modern integrated development environments for languages such as Java, C#, F# or Scala. Therefore, it is impossible to truly compare the design-time support for this aspect of expressiveness. However, since dependent type providers are simply generating ordinary terms, we expect that the “design-time” support for them will be similar to that for other Idris programs.

5. Elaborating Idris Type Providers

Idris’s semantics is defined by translation to a simple core language, referred to as \( \text{T} \), in a process known as elaboration. The core language is a straightforward dependently typed \( \lambda \)-calculus with top-level pattern-matching definitions and inductive families. The details of this elaboration process are explained in Brady [5].

As an example, consider the elaboration of Idris’s case construct, which is similar to Haskell’s. There is no pattern-matching case expression in \( \text{T} \) — all pattern-matching must occur in top-level definitions. The elaboration process converts a case construct into a new top-level pattern-matching definition and inserts a call to the newly-defined function in place of the original case expression.

Take \( \text{IO} \) and \( \text{Provider} \) to be the \( \text{T} \) types declared in Figure 2. In the actual implementation, \( \text{Provider} \) is elaborated from an ordinary Idris datatype, while \( \text{IO} \) is treated specially by the compiler. In particular, \( \text{IOCon} \) has no user-accessible name in Idris. Brady [5] defines a monadic tactic language, in the spirit of Coq, in which he can specify the nitty-gritty details of elaborating the highly implicit Idris into the completely explicit \( \text{T} \). The paper has a highly operational approach, demonstrating monadic tactic scripts to elaborate each feature. Because the elaboration process for type providers is much more straightforward than features such as implicit arguments, this presentation does not need the full power of the tactic language. Additionally, explaining the semantics of Brady’s tactic language would be quite involved. Thus, we present the elaboration process using inference rules. Figure 1 demonstrates the judgments used to represent the process of elaboration.

The typing relation has the form \( C; \Gamma \vdash t_1 : t_2 \). This represents that \( t_1 \) has type \( t_2 \) in the global context \( C \) and the local context \( \Gamma \). Terms at the top level are typed in the empty local context, but they have access to earlier global definitions. However, the elaboration process for terms can cause variables in the local context to be solved by unification — thus, the process creates a new local context.

The judgment \( C; \Gamma \vdash t_1 \xrightarrow{\text{exec}} t_2 : T \) states that, in the global context \( C \) and local context \( \Gamma \), executing the term \( t_1 \) yields the result term \( t_2 \). Both terms are expected to have type \( T \).
The judgment $C; \Gamma \vdash \ell : \text{elab} \to \text{Ok} \left( \tau ; \Gamma' ; d'_{\ell} \right)$ states that in the global context $C$ and the local context $\Gamma$, the Idris term $\ell$ elaborates to the $\ell$ term $\ell_{\Gamma'}$, producing a new local context $\Gamma'$ and the possibly-empty sequence of new top-level definitions $d'_{\ell}$. The corresponding error judgment $C; \Gamma \vdash \ell : \text{elab} \to \text{Fail} \ \text{err}$ indicates that elaboration of $\ell$ fails with some error $\text{err}$.

The judgment $C; \Gamma \vdash d_{\ell} : \text{elabD} \to \text{Ok} \left( d'_{\ell} \right)$ indicates that the Idris top-level definition $d_{\ell}$ is successfully elaborated to the sequence of $\ell$ definitions $d'_{\ell}$. The corresponding error judgment $C; \Gamma \vdash d_{\ell} : \text{elabD} \to \text{Fail} \ \text{err}$ indicates that elaboration of the Idris definition $d_{\ell}$ fails with some error $\text{err}$.

This technique places very few requirements on the specific semantics of the execution relation in order for it to make sense as a component of a dependent type providers system. The current presentation assumes the existence of some constructor $\text{IOCon}$ from which the final value can be extracted. However, dependent type providers need only a way to distinguish fully-executed terms from not-fully-executed terms and a means of extracting the result of execution. Additionally, dependent type providers require that execution preserves types, though progress is not essential. If execution does not produce a value, typechecking can simply be aborted. In practice, however, the execution relation should be the same one implemented for the host language at runtime.

Figure 3 describes the process of successful type provider elaboration. First, the declared type is elaborated to some $\ell_T$ term $\tau$. Assuming that $\tau$ is in fact a type, the provider term $p$ is then elaborated to a $\ell_T$ term $\pi$. If the elaborated provider term has the appropriate type $\ell$ ($\text{Provider} \ \tau$), then it is executed, and if execution terminates normally, the resulting term is inspected. If the resulting term was $\text{Provide} \ t$, then the entire $\text{Provide}$ clause is elaborated to the declaration $x : \tau$ and the definition $x = t$, along with any auxiliary definitions $d_1$ and $d_2$ that resulted from elaborating the type and the term.

The new local contexts $\Gamma_1$ and $\Gamma_2$ that are produced through elaboration in Figure 3 are thrown away. This is intentional. However, the final resulting $\ell_T$ terms must again typecheck in only the global context.

If the resulting term was $\text{Error} \ \text{err}$, then elaboration of the program fails, reporting the error $\text{err}$. This error elaboration is defined in Figure 3. In this way, type providers can signal errors that convey useful information to users.

### 6. Example: CSV Type Provider

In this section, we describe a simple type provider for comma-separated value (CSV) files. Like the F# CSV type provider [16, pp. 42–43], this provider inspects an example of the particular CSV format that it should read at compile time. The type provider treats the first line of the file as column names, allowing users to look up data using these names and statically checking that the employed names exist.

The F# CSV type provider additionally infers column types from the form of the data in each column. Users can also provide type annotations in column headers. For the sake of simplicity, the example in this section does not have these features, but they would be straightforward to implement using a technique similar to the database type provider discussed in Section 7.

There is no entirely standard definition of CSV file. Here, we take it to mean a plain text file where each line represents a record. Each record consists of the same number of fields, separated by some delimiter character. This delimiter character is often a comma, but it need not be. In our dialect of CSV, every field has a name, and the first line of the file specifies the names of each field.

#### 6.1 Named Vectors

The first step in designing a dependent type provider is to design a datatype that can represent the schema in question. In the case of our CSV type providers, we want to ensure that numeric column references are within bounds and that lookups by name use names that are defined in the first line of the file. This can be represented using the datatype $\text{NamedVect}$.

```
data NamedVect : Type -> (n : Nat) -> (Vect String n) -> Type where
  Nil : NamedVect a 0 []
  (::) : a -> NamedVect a n ss -> NamedVect a (S n) (s :: ss)
```

Note that $a$, $n$, $s$, and $ss$ are all implicit arguments, because they begin with a lower-case letter and are not otherwise bound. Additionally, due to Idris’s overloaded list literal syntax (see Section 2.2), one can write:
test : NamedVect String 2 ["name", "age"]
test = ["George", "45"]

This datatype is similar to Vect, a traditional introductory example used with dependent types. A Vect a n is an n-element list whose elements are of type a. The type NamedVect has an additional index: a Vect that must be the same length, containing strings that indicate column names.

While Idris does feature record types, they are simply syntactic sugar for ordinary datatype definitions with special accessor and update functions. Thus, record types cannot be the result of a computation. Therefore, type providers for record-like data must imitate record types using Idris’s expressive type system. In some sense, a NamedVect is a kind of restricted record type in which all fields have the same type. This approach can be extended to encompass other, richer emulations of record types. One model for this can be seen in the representation of database schemas in Oury and Swierstra [15].

Numeric lookup is statically checked using the type Fin, where Fin n is a type that has precisely n elements. Therefore, an element of Fin n serves as a bounds-checked index into a NamedVect a n ss. The datatype Fin is defined as follows:

data Fin : Nat -> Type where
  f0 : Fin (S k)
  fS : Fin k -> Fin (S k)

Note that it is impossible to construct an instance of Fin 0, as both constructors apply the successor constructor S in their indices. The function index implements a safe numeric lookup:

index : Fin n -> NamedVect a n ss -> a
index f0 (x :: xs) = x
index (fS f) (x :: xs) = index f xs

No case is necessary for the empty NamedVect because it is impossible to construct an instance of Fin 0, as and the second argument were the empty NamedVect then n would be 0.

Named-based lookup is somewhat more interesting. The datatype Elem ss s represents a proof that the string s is an element of the vector ss. This type has two constructors: Here takes a proof that s is the head of ss, returning a proof of Elem ss s, and There takes a proof that s is in the tail of ss. This datatype happens to have a similar structure to natural numbers. Thus, it can be used to look up the result:

lookup' : (s : String) ->
  NamedVect a n ss ->
  Elem ss s ->
  a
lookup' s (x::xs) (Here _) = x
lookup' s (x::xs) (There rest) = lookup' s xs rest
lookup' s [] pf = elemEmpty pf

What remains is a convenient interface to users that finds the Elem proof, or fails to compile. The implicit argument x is declared auto, which means that it will be filled out using refl, the constructor for the equality type. This will trigger the decision procedure decElem while performing unification with Yes pf, where pf refers the implicit argument. If decElem returns No, it is a static error.

lookup : (s : String) ->
  NamedVect a n ss ->
  {prf : Elem ss s} ->
  {auto x : decElem ss s = Yes prf} ->
  a
lookup s nv {prf=p} = lookup' s nv p

6.2 Representing CSV Types

A simple universe is used to represent CSV types. To actually parse a CSV file, one must know both the delimiter character and the number of columns that are expected (in order to handle malformed rows). To construct the named vector type, one must additionally know the names of the columns. The datatype CSVType is a simple representation of this information, and the function Row interprets an element of CSVType as the type of a row from the corresponding CSV file.

data CSVType : Type where
  MkCSVType : (delim : Char) ->
  (n : Nat) ->
  (header : Vect String n) ->
  CSVType

Row : CSVType -> Type
Row (MkCSVType d n h) = NamedVect String n h

6.3 The Type Provider

The type provider is a means of inferring a CSVType from a CSV file. To make it useful, there is also a collection of generic functions that map codes in the CSV universe (that is, elements of CSVType) to concrete functions that work on a specific CSV type.

The function inferCSVType takes a delimiter character and the first line of a CSV file as an argument. It then splits the line using the delimiter character and uses the result to construct a CSVType:

inferCSVType : Char -> String -> CSVType
inferCSVType delim header =
  let cs = cols delim header
  in MkCSVType delim (length cs) (fromList cs)

The cols function simply splits the header string where the delimiter character occurs, trimming the leading and tailing whitespace around each substring. Our type provider csvType is a thin wrapper around inferCSVType that ensures that the input file has at least one line:

csvType : Char -> String -> IO (Provider CSVType)
csvType delim filename =
  do lines <- readLines filename
      return $
      case lines of
        [] => Error ("Could not read header of " ++ filename)
        (h :: _) => Provide $ inferCSVType delim h

The CSV type provider is used with the %provide construct:

%provide (t : CSVType) with csvType ; "test.csv"

The contents of the file test.csv can be seen in Figure 5.

The function getAge is a statically checked accessor that uses the function lookup together with the name of the "Age" column to retrieve its contents:

getAge : Row t -> String
getAge r = NamedVect.lookup "Age" r

Note that the signature of getAge refers to t, the provided CSV type. If the "Age" column in test.csv is renamed, then the module will no longer typecheck.

Finally, the provided type can be used. The function main is an IO action that reads a CSV file and displays the contents of two of the columns. The function readCSVFile simply reads a file according to a CSVType, in this case t, throwing away the header row and ignoring rows that are not well-formed.
8. Alternate Designs

One might wonder whether the presented design for type providers could either be more general or safer. This chapter explores the design space for type providers in a dependently-typed language. A number of alternative formulations of the feature are discussed and analyzed, and each is found wanting.

8.1 Top Level vs Expression Level

Dependent type providers can only be used to generate top-level definitions. For example, it is not allowed to restrict the scope of a provided type using a let binding, as in the following hypothetical code snippet:

```haskell
let db = provide (loadSchema "some database")
```

While this limitation can to some extent be worked around by simply referring to a top-level definition that is the result of a type provider, it might nevertheless seem like an arbitrary restriction.

However, there is a good reason for this limitation. If type providers could be used in arbitrary expression contexts, then they could contain free variables whose values will not be known until runtime. For example, our hypothetical database type provider might be used in the body of a function that receives the database name as an argument:

```haskell
getStuff thing database =
  let db = provide (loadSchema database)
  in query db (mkQuery thing)
```

There are two reasonable choices for the semantics of expression-level type providers: either every provider with free variables will be executed at runtime, when the values will certainly be known, or it should be a static error to refer to dynamic values. The former choice is not particularly satisfying, as the type provider mechanism would simply be equivalent to Haskell’s `unsafePerformIO`, due to the extraction of the result term from the IO context.

On the other hand, the latter choice is also not particularly satisfying. It should be possible to perform a binding-time analysis [12] to determine whether a free variable in a type provider expression is statically known. However, this may be difficult for users to understand, and it may lead to an inability to predict whether an uncontrollable side effect will occur at compile time or when the program is run.

Requiring that invocations of type providers occur at the top level is thus a significantly simplifying assumption for both the implementation and for users, with only a minor practical cost.

8.2 Unrestricted I/O

A simpler means to achieve the goals of Idris’s type providers would be to extend the Idris evaluator to support an analog of Haskell’s `unsafePerformIO`. This would be strictly more powerful than dependent type providers.

However, this approach suffers from two major drawbacks. First, it might be difficult to predict exactly when or even how many times an effect will be executed, which could lead to unpredictable results. Predicting the behavior of an effectful term would require a deep knowledge of the type checker’s internal algorithms. Second, it makes it possible for compile-time effects to “hide” in other code, with users unaware that the effects are occurring. Type checking a file could have any effect, and it would be very difficult to discover which effects type checking a particular file might invoke. The top-level type provider syntax `%provide` resembles a compiler directive, which signals that something potentially dangerous will occur.

8.3 Restricted I/O

As an alternative to allowing unrestricted actions in the IO monad, it might instead be possible to create a restricted, domain-specific language in which type providers could be defined. The key insight...
is that input is much more important that output. If a type provider can be prevented from sending angry email to your boss or deleting all your files, while retaining its ability to read database schemas, then the feature becomes much safer to use.

In practice, however, this would undermine the usefulness of type providers. The primary purpose of type providers is to interact with data sources that are defined outside of Idris. In practice, this will probably involve linking to libraries written in C. Linking to C libraries automatically invalidates all safety guarantees, including memory safety, and C code can certainly perform arbitrary effects.

The alternatives are to reimplement large swaths of code in the restricted input language, which may not even be possible, or to extend the language anew for each new data source.

For these reasons, it seems unlikely that a restricted, safe DSL for defining type providers will be particularly useful. Type providers are useful precisely because they can do almost anything to construct a model of something outside of the language.

9. Conclusion

We have demonstrated dependent type providers, a language extension that allows many of the benefits of F#'s type providers while preserving the safety properties of the type system. We have implemented dependent type providers in the Idris programming language.

Dependent type providers in Idris have some advantages over F#'s type providers. Because they work entirely with ordinary terms, there is no risk of violating the assumptions of the type system, while F#'s type providers can use their access to the internals of the compiler to generate illegal types and ill-typed terms. The straightforward semantics of the dependent type provider mechanism means that they can be developed in the same manner as any other Idris program and that they can be defined and used in the same module.

There are also some key disadvantages of the present Idris implementation of dependent type providers relative to F# type providers. The lack of laziness means that it would not be practical to use Idris type providers to access very large schemas, such as Freebase. Additionally, because dependent type providers are not able to introduce new identifiers to the global context, they must resort to naming attributes of external data with strings, relying on compile-time proof search to ensure safety. This was demonstrated in Section 6. It would be difficult to make this arrangement as convenient for end users as introduced identifiers.

Other aspects of the two systems are not entirely comparable. It is unclear to what extent generated .NET types can represent more or less interesting invariants than indexed datatypes, especially when the usual well-formedness requirements are lifted. For example, it is straightforward to define an Idris type to represent SQL's VARCHAR(n) using a variant of Vect that uses its Nat index in a fashion similar to Fin. It is not immediately clear how to do this using .NET types. On the other hand, .NET subtyping is very expressive, and may provide opportunities that dependently-typed functional languages lack.

Dependent type providers have a great deal of potential that has not yet been realized. Future work will focus on removing restrictions as well as discovering new applications.

10. Future Work

10.1 Laziness and Erasure

Dependent type providers currently lack one of the key features of F# providers: the ability to generate types lazily, on an as-needed basis. As such, it would not be practical at this time to write a type provider for Freebase with its more than 23,000 types. The resulting terms inside of the typechecker would probably slow the system down to the point of impracticality.

Idris, however, already has optional lazy evaluation. As currently defined, the executor forces every thunk when returning a term from a type provider. This means that laziness annotations can be practical during the execution of a type provider, but they are useless in terms that are returned from a type provider. It would be interesting to explore the use of Idris's already-existing support for lazy evaluation for generating type providers lazily, to see if this is sufficient for the purposes of laziness in F# type providers. This will require modifications to the trusted core evaluator in Idris, as side-effectful execution will need to be interleaved with the reductions performed during typechecking.

Likewise, F#'s ability to replace provided types with a less-specific supertype in compiled code is key to avoiding a code size explosion in the presence of very large external data schemas. Idris already includes aggressive type and term erasure, based on the techniques described in Brady et al. [8] and Brady [6]. It would be instructive to explore to what extent these optimizations apply to uses of type providers, and what techniques are necessary to trigger them.

10.2 Proof Providers

Idris supports reflection of terms, converting Idris terms to an abstract syntax tree of the corresponding TT term and back again. This is similar to the corresponding feature in Agda [19]. It would be interesting to extend dependent type providers such that they can receive a reflected representation of the goal type as an argument. Then, it would be a mere matter of programming to write a provider that attempts to use an external proof search tool (e.g. Z3 [14]) to either find a term that inhabits the goal type or, perhaps more realistically due to the differences in logics between proof tools, convert the type into a postulate.

10.3 Foreign Function Interfaces

While Haskell’s foreign function interface (FFI) to C requires declarations of foreign C types, Idris’s C FFI uses a universe representing a subset of C types to declare type information. Thus, it should be straightforward to parse a C header file and extract Idris declarations for every function.

Additionally, a similar mechanism could be used to automatically generate interface declarations for a variety of foreign interfaces. For example, one could parse JVM or .NET bytecode and generate a type-safe interface. Finally, this mechanism could be used to read the interface of other foreign code, like web service definitions in WSDL.

10.4 Runtime Value Providers

Type providers create values that are constant at compile time. However, some other values are constant throughout a particular execution of a program, but they cannot be statically determined. Examples include the values of environment variables, the current locale, the system time zone, and the command line used to execute the program. All of these values are the results of I/O actions, but they are also constant throughout the execution of a program.

A straightforward extension of the type provider mechanism would consist of a separate declaration that caused the type provider to be executed at runtime, but not at compile time. The type checker would need to be extended to prevent unexecuted runtime value providers from being unified with anything to prevent them from being used in dependent types, as they are by definition not known at compile time.

This could be a convenient way to formulate libraries that need to perform input or output. It corresponds roughly to the Haskell idiom of defining a top-level value with unsafePerformIO, though
with better guarantees for when the action is executed, an error handling mechanism, and the ability to use tools developed for type providers.

11. Related Work

There are a large number of related techniques. F# type providers, being the inspiration for dependent type providers, are the most obvious related work.

Syme et al. [16] provide a thorough description of the related work on type providers in general. Rather than repeat the final section of their report, this chapter discusses particular related work to the approach taken with dependent type providers.

11.1 Generic Programming with Universes


Dependent type providers use the techniques of generic programming with universes to accomplish a goal that is perhaps the opposite of generic programming. While generic programming seeks to define operations in a datatypes-agnostic manner to achieve greater generality, dependent type providers seek to restrict the allowed programs to those that will work in a particular environment.

11.2 Ur/Web

Ur [9] is a functional programming language that was designed to support a Web programming framework called Ur/Web. Ur has a highly expressive type system that allows metaprogramming with record types. This system is rich enough to support static HTML and database query validation. Despite this expressiveness, Ur does not support dependent types. Type-level programming occurs in a separate class of terms.

The type system is carefully designed to allow type inference where metaprograms are invoked and to ensure that users almost never have to write a proof term. While it is certainly possible to write the kinds of metaprogams that one can write in Ur in a language with full dependent types, it is probably not nearly as convenient.

Presently, Ur/Web does not have a type provider system. However, as Syme et al. [16] point out, such a mechanism would likely be pleasant to use. The lack of dependent types in Ur means that Idris-style type providers would not be practical, so it would need to resemble something like F#’s type providers.

Ur/Web’s record metaprogramming falls in a “sweet spot,” where many interesting properties can be described in a very convenient system. It would be particularly interesting to see if a similarly convenient system could be implemented as an embedded domain-specific language in Idris and then used by type providers.

11.3 Metaprogramming Dependent Types

A common theme in dependently-typed programming is that one often wishes to have the machine generate a part of a program. Typically, this is the part corresponding to a proof of some property. We saw this in both example type providers, where proof automation was used to conveniently enforce properties such as the presence of a field name in a structure or the disjointness of two schemas. This kind of metaprogramming is a vast field. Here, we simply consider the potential of such features as type provider mechanisms.

The Coq system [18] uses metaprograms in the Ltac language to create proof terms. Adding type providers to Coq could be as simple as writing a tactic that creates terms based on an external data source.

The programming language Agda [17] contains a reflection mechanism, described in van der Walt’s M.Sc. thesis [19], in which an Agda program can create abstract syntax trees for Agda terms. The resulting terms are type-checked just as any other Agda terms. While Agda could certainly support dependent type providers as described in this paper, F#-style type providers in Agda could be implemented by allowing these terms to be produced by arbitrary I/O actions. These type providers would be in the tradition of F# because they rely fundamentally on code generation rather than running ordinary programs for their effects.

Acknowledgments

This work was performed as part of the Actulus project, which is generously supported by the Danish Advanced Technology Foundation (017-2010-3). I would like to thank Hannes Mehnert and my advisor Peter Sestoft for feedback on drafts of this paper. Additionally, I would like to thank Hannes, Peter, Nicolas Pouillard, and Daniel Gustafsson for enlightening discussions about dependent type providers and Edwin Brady for his assistance with the Idris implementation. Last but not least, I would like to thank the anonymous reviewers for their detailed and helpful feedback.

References


