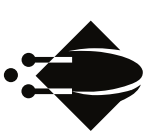
Looking Outward: When Dependent Types Meet I/O

David Raymond Christiansen

M.Sc. Thesis

Supervisor: Peter Sestoft
Submitted: June 10, 2013
(corrected version)



IT University
of Copenhagen

Abstract

Type providers, 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 real-world data. 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 one can define a type provider mechanism that does not rely on code generation. This mechanism uses the ordinary abstraction mechanisms of the type system to represent the schemas of external data sources. We evaluate the practicality of this technique and explore future extensions.

Resumé

Programmeringssproget F# har en ny feature, kaldet *Type Providers*, som gør det muligt, at udnytte de fordele, som hører til moderne statiske typesystemer, når man arbejder med data fra eksterne systemer. *Type providers* implementeres i F# ved at bruge en slags kodegenerering ved oversættelse, hvilket kræver indgreb i F#-implementationen og kan underminere sædvanlige typegarantier. Vi demonstrerer, at *dependent types* gør det muligt at definere en mekanisme til *type providers*, som ikke er baseret på kodegenerering. Denne mekanisme anvender typesystemets almindelige abstraktionsværktøjer til at repræsentere eksterne datakilders skemaer. Vi evaluerer teknikkens anvendelighed og undersøger fremtidige udvidelser.

Acknowledgments

I would like to thank my supervisor, Peter Sestoft, for his intensive help with reviewing this thesis, for interesting discussions, and for frequent infusions of knowledge and communication skills. Additionally, I received a lot of help during the process from Edwin Brady, who was always willing to answer silly questions about the internal architecture of the Idris compiler. Thank you, Edwin.

Thank you to Hannes Mehnert, Daniel Gustafsson, and Nicolas Pouillard for enlightening discussions and for keeping one of my feet firmly planted in the ivory tower, and thank you to the people at Edlund A/S for keeping the other foot outside.

Thank you to my proofreaders, Jeanne Stevenson, Mads Bjerregaard Andersen, Kasper Videbæk, Nicolai Dahl Blicher-Petersen, and Adam and Maria Britt for useful comments and suggestions.

Thank you, *Højteknologifonden* and ITU, for sponsoring this work and putting a roof over my head for the past two years. In particular, my work as part of the Actulus project is sponsored by *Højteknologifonden* grant number 017-2010-3.

And thank you Lisbet, for your graphical eye and for your continuing patience with the demands of university life.

Contents

Introduction

Dependent types, in which types can be parameterized by values, are in some sense the "final frontier" of statically-typed programming. In the past decade, features such as polymorphic types that could previously only be found in languages from the academic community have found their way to mainstream languages such as C#, Java, and Scala. Additionally, industry giants such as Microsoft have begun supporting functional programming, distributing a dialect of ML called F# with their popular integrated development environment Visual Studio.

Access to the last face of Barendregt's lambda cube [Bar91] has been jealously guarded by the demons of impracticality. In some earlier systems, it was so difficult to program with dependent types that they were only used separately as a logic to prove properties of programs written in the non-dependently-typed subset of the language. However, work on languages such as Epigram [MM04], Agda [Agda], and Idris [Bra11] has been bringing us steadily closer to useful models for using dependent types in our programs. This thesis describes another small step along this path.

F# 3.0 includes a feature called *type providers*, which enables statically-typed access to externally-defined data sources. Type providers allow the full range of type-based tools in F#, including automatic completion of identifiers in the IDE, to be applied to data sources as diverse as relational databases and Twitter feeds. These type providers are implemented using a special API in the F# compiler that allows both types and code to be generated. While this feature is fantastically useful, it is in some sense not completely satisfying. The process of defining a new type provider breaks the abstraction of the language, showing too much of what is happening "behind the scenes".

With dependent types, we can do better. Code generation is necessary for type providers on the .NET platform because it can only represent facts about external data sources as *data*, but it needs to use *types*

to statically check properties of our data. However, dependent types already allow us to use data to control our types. A straightforward language extension allows datatype indices to be produced by ordinary programs.

The main contribution of this thesis is a language extension called *Idris type providers*, that we have implemented in the Idris compiler. We describe their background and inspiration, design, semantics, and implementation. We compare their expressive power and convenience to F# type providers, both through theoretical discussion and by implementing two example type providers.

A type provider mechanism is a success to the extent that:

1. It is possible to derive a *safe* and *convenient* API from the schema of an external data source
2. It *supports very large schemas* in said external data sources
3. It is *easy to use*

We determine that Idris type providers support goals (1) and (3), and that a future extension could enable (2) as well.

It is intended that readers who are skilled users of Haskell or another functional programming language but who do not necessarily have a background in dependent types or F# will be able to understand this work. Thus, significant space is used on explaining the necessary parts of dependent types as they are used in Idris.

Part I describes the Actulus project, the applied context within which this work was performed. The project uses advanced programming language technology to create the next generation of life insurance and pension management systems.

Part II describes the background information that is necessary to understand this work, in particular F# type providers and dependently-typed functional programming.

Part III describes the technique itself and describes potential alternatives.

Part IV evaluates the technique in relation to the goals of type providers, both practically and theoretically.

Part I

Project Context

Chapter 1

Actuarial Models and Actulus*

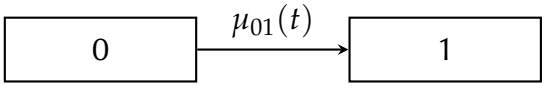
Computational demands on life insurance and pension funds are increasing. In the wake of the financial crisis of 2008, the European Union issued a new directive, called *Solvency II* [SII09], that intends to minimize the risk of systemic insurance and pension fund collapse. Living up to these requirements poses new challenges that require significant changes to actuarial software infrastructures. The Actulus project, a collaboration between the University of Copenhagen, Edlund A/S and the IT University of Copenhagen, seeks to solve this problem through a combination of advances in actuarial science and programming language research, taking advantage of Edlund's position as a market-leading vendor of software to the life insurance and pension industry in Denmark.

A key component of Actulus is the Actulus Modeling Language, or AML. AML is a statically typed, total functional language. It is described in Chapter 2. This chapter motivates the design of AML by presenting the fundamentals of the actuarial models used in Actulus.

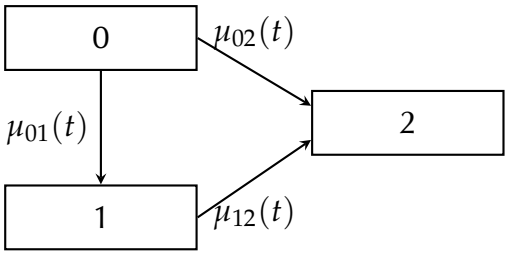
Actulus is concerned with life insurance and pension products. These products are traditionally represented inside of state models, with states corresponding to the condition of the insured, such as able-bodied, disabled, or dead, and transitions that indicate the possible changes of state. While some actuarial traditions and systems represent these discretely, the Danish tradition is to use continuous-time models. In particular, products are represented using continuous-time Markov models.

A continuous-time Markov model consists of a finite number of *states*,

*This chapter is adapted from part of a paper submitted to the 30th International Congress of Actuaries, 2014 [Chr+14]



(a) Two-state mortality model



(b) Three-state disability model

Figure 1.1: State models

typically denoted by the numbers $0, 1, 2, \ldots$ and *transition intensities* between these states. The transition intensity $\mu_{ij}(t)$ from state i to state j, when integrated over a time interval, gives the probability that a transition from state i to state j will occur in the interval. These models exhibit the *Markov property*, which is to say that future transitions depend on the past only through the current state. Some products are much easier to define using *semi-Markov models*, in which the transition intensities can additionally depend on the duration of sojourn in the source state, but these models are not the present focus of work on Actulus.

A simple two-state Markov model, as seen in Figure 1.1(a), can be used to represent the mortality of a single person. State 0 represents that the insured is alive, while state 1 represents that he or she is dead. The mortality intensity $\mu_{01}(t)$ represents the rate of mortality for the insured, which will be determined from information such as the age, sex, and occupation of the insured.

A slightly more complicated model must be used for products offering disability insurance. These products can be modeled with three states, representing active labor market participation, disability that precludes employment, and death, respectively represented as 0, 1, and 2 (Figure 1.1(b)). There are transitions from active participation to disability and to death, and from disability to death. Additionally, some

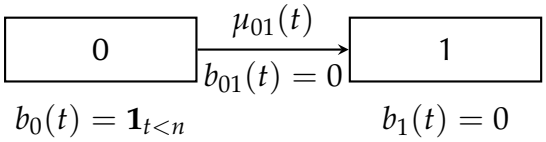
products may allow for *reactivation*, where a previously disabled customer begins active labor again. Many current actuarial systems are restricted to acyclic models. Because the AML system uses a numerical differential equation solver, it can handle cyclic models.

Life insurance and pension products are modeled by identifying states in a Markov model (of the life of the insured), by attaching payment intensities to the states, and by attaching lump-sum payments to the transitions. We use $b_i(t)$ to denote the payment stream in state i and $b_{ij}(t)$ to denote the lump sum due on transition from i to j at time t.

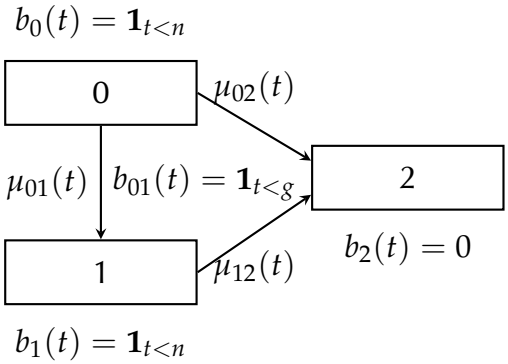
An example product in a two-state model is the temporary life annuity, where repeated payments are made to the policy holder until some expiration date n, provided that he or she is alive. Our model contains two states: 0, in which the policy holder is alive, and 1, in which the policy holder is dead. Obviously, there is just a single transition: from 0 to 1. We have some mortality intensity $\mu_{01}(t)$, and because no payment is to occur at or following death, we know that $b_{01}(t) = 0$ and $b_1(t) = 0$. Using the syntax $\mathbf{1}_\phi$ to represent a function that returns 1 just in case ϕ holds, and 0 otherwise, we have $b_0(t) = \mathbf{1}_{t<n}$. That is, the policy pays a constant stream of one unit of currency per unit of time until $t \geq n$ or until the policy holder dies. Figure 1.2(a) shows this product.

We can extend our temporary life annuity with a disability sum. A disability sum pays a lump sum when the policy holder is declared unfit to work prior to some expiration g. For this task, we use a three-state model where state 0 represents active labor-market participation, state 1 represents disability, and state 2 represents death. Clearly, we have transitions from 0 to 1, from 1 to 2, and from 0 to 2. It would be possible to have a transition from 1 to 0, representing the policy holder returning to the labor market after a period of disability, but we will assume that this product pays its disability benefit at most once. Because both able-bodied and disabled holders are alive, we have $b_0(t) = b_1(t) = \mathbf{1}_{t<n}$, just as in our original product. Additionally, assuming that the disability sum is one unit of currency, we have $b_{01}(t) = \mathbf{1}_{t<g}$. Figure 1.2(b) shows the extended product.

To avoid bankruptcy, insurers must maintain enough assets to pay for their obligations to the insured. In other words, they must maintain a reserve of capital. The *prospective statewise reserve* $V_j(t)$ is the reserve at time t given that the insured is in state j at time t. This is the expected net present value at time t of future payments in the product, given that the insured is in state j at that time, and given information up to time



(a) Temporary life annuity



(b) Temporary life annuity with disability sum. Both $b_{02}(t) = 0$ and $b_{12}(t) = 0$ are omitted for reasons of space.

Figure 1.2: Products

t. The *principle of equivalence* states that the reserves at the beginning of the product should be 0 — that is, the expected premiums should equal the expected benefits.

We compute the prospective statewise reserves using Thiele's differential equations (see Figure 1.3), a system of ordinary differential equations with one equation for each state in the product model. The specific details of Thiele's differential equations are beyond the scope of this chapter. It suffices to note that the various parameters are all drawn directly from the product model. Additionally, analytic solutions are only available for a subset of products. At the very least, the presence of cycles in the transitions of a state model means that Thiele's equations do not have closed-form solutions. Insurers whose software systems rely on analytic solutions must either use wildly unrealistic models or simply not offer certain products.

Conceptually, there is also a certain inelegance in this manner of

$$\frac{d}{dt}V_j(t) = \left(r(t) + \sum_{k;k\neq j} \mu_{jk}(t)\right) V_j(t) - \sum_{k;k\neq j} \mu_{jk}(t)V_k(t) - b_j(t) - \sum_{k;k\neq j} b_{jk}(t)\mu_{jk}(t)$$

where
$V_j(t)$ is the statewise reserve for state j at time t
$r(t)$ is the interest rate at t
$\mu_{jk}(t)$ is the transition intensity from j to k at t
$b_j(t)$ is the payment intensity in state j at t
$b_{jk}(t)$ is the lump-sum payment due on transition from j to k at t

Figure 1.3: Thiele's differential equations

defining products. Presumably, the intensity of mortality is related to factors such as age, sex, and lifestyle. These factors are independent of the particular insurance product. Likewise, a pension plan offered to two different customers will still have the same structure. Furthermore, one customer's life insurance plan is unlikely to have an effect on the interest rate. The AML language, described in the next chapter, attempts to address these issues using a combination of numerical differential equation solvers and programming language techniques.

Chapter 2

The Actulus Modeling Language

In Chapter 1, we saw that traditional actuarial models mix up information about the probabilities of certain events in the life of the insured (*i.e.* the transition intensities) with information about the contract between the insurer and the insured (*i.e.* the payments). This leads to a duplication of information between products models, both when modeling multiple products for one customer and multiple customers with the same product. To make matters worse, Thiele's differential equations even add projections about the future of society (*i.e.* the interest rate model) to this mix. These three components need not be defined together.

We have developed AML in close collaboration with Niels Kokholm, Henning Niss, Klaus Grue and Kristján Sigtryggsson from Edlund A/S and with Peter Sestoft from the IT University of Copenhagen. Large parts of the language design are due to the IT University. It is a part of the Actulus project,[1] an interdisciplinary research collaboration between the University of Copenhagen, Edlund A/S, and the IT University of Copenhagen. This chapter describes work in progress. We have not yet worked out all of the details of the AML language, but the general outline is in place. We focus on the parts of AML that relate to the actuarial domain, as the language is in many ways an ordinary functional programming language.

AML is intended to be a total functional language *à la* Turner [Tur04]. Because it is a domain-specific language where all programs use finite data and should terminate, we do not intend to include facilities for ei-

[1]Supported by the Danish Advanced Technology Foundation *(Højteknologifonden)* (017-2010-3)

```
statemodel LifeDeath(p : Person) where
  states =
    alive
    dead
  transitions =
    alive -> dead
```

Listing 2.1: A state model for tracking mortality

ther codata or infinite or cyclic data structures. Totality will help ensure that the ODE solver doesn't attempt to evaluate non-terminating functions, and it will hopefully enable useful analyses and optimizations.

2.1 Introduction to AML

AML separates the product definition into two components: a *risk model* consisting of the transition intensities and a *product* consisting of the payment streams and lump-sum payments. We use the term *state model* to refer to the collection of states and transitions that are available in a given Markov model, as separate from the risk model and the product. A risk model and product can be combined if they are defined within the same state model.

The AML definition of the two-state mortality model from Figure 1.1(a) can be seen in Listing 2.1. State models may take parameters; however, there is no meaningful way to use these parameters in the body of the state model. State model parameters will be important in Section 2.2.

Indented code blocks are syntactically meaningful in AML. Listing 2.2 defines the temporary life annuity in this state model. Or, more accurately, it defines a family of temporary life annuities that vary based on the provided expiration date `n`. The construct `at t` binds the variable `t` to the present time in the remainder of the payment specification. The construct `pay ¤1 per year` creates a payment stream with a constant payment intensity of one unit of currency per year,[2] and `provided` serves the dual function of attaching a payment stream to a state and imposing the conditions that come from the $\mathbf{1}_{t<n}$ in Figure 1.2(a).

[2] '¤' is the standard symbol for an unknown or unspecified currency

```
product TempLifeAnnuity(p : Person,
                        n : TimePoint) : LifeDeath(p) where
  benefits = at t pay ¤1 per year
                  provided (alive and t < n)
```

Listing 2.2: Temporary life annuity

```
riskmodel RiskLifeDeath(p : Person) : LifeDeath(p) where
  intensities = alive -> dead by gompertzMakehamDeath(p)
```

Listing 2.3: Risk model

Listing 2.3 demonstrates a risk model for `LifeDeath`. Risk models must specify an intensity for each transition. The type `Person` represents a record type with information such as name, sex, and date of birth. The function `gompertzMakehamDeath` simply generates a transition intensity function from a `Person`.

In AML, a *calculation basis* for some product consists of a risk model that matches the product's state model, a model of the interest rate $r(t)$, and any additional information that might be necessary to construct the differential equations for the product. Additionally, calculation bases include a cutoff date that is necessary for the differential equation solver. For example, calculating the reserve of a product that pays a benefit to any surviving children upon the death of the insured requires a model of the likelihood that the insured has children.

2.2 Static Types for AML

AML is a statically-typed functional programming language with at least some degree of dependent types. Further evaluations of usability by real actuaries will guide the extent to which full dependent types are exposed to users. We do not describe the basics of the AML type system here, because it is substantially similar to other functional programming languages and because the particular details that might distinguish it from other systems have not yet been firmly established. This section explains the unique features of AML's type system as they relate to the actuarial computations described in Chapter 1.

```
value model : StateModel = LifeDeath(anne)

value product : Product(model) =
  TempLifeAnnuity(anne, DateTime(2035,1,1))

value risk : RiskModel(model) =
  RiskLifeDeath(tony) -- Type error
```

Listing 2.4: Mismatching dependent state models

2.2.1 State Model Types

While state models in Section 2.1 resemble types syntactically, they are in fact data. There exist types `Product(SM)` and `RiskModel(SM)` that are the respective types of products and risk models for some state model `SM`.

In most cases, we expect that state models will be parameterized over the person whose life is being measured. This is because AML features dependently-typed product definitions. That is, arguments to products or risk models can occur as parameters to the state model that they inhabit. Two state models are only equal if they have equal arguments. By including the person whose life is being measured as a parameter to the state model, we can distinguish risk models and products for one person from those for another.

Let `tony` and `anne` be `Person` instances representing a 70-year-old man and a 23-year-old woman, respectively. Presumably, Anne will live significantly longer than Tony. Clearly, it would be a mistake to calculate the reserves for Anne's temporary life annuity using a model for Tony's mortality. The overestimation of the probability of death would lead to the reserve, and thus the premium, being too low.

Because the state model `LifeDeath` takes a `Person` parameter, however, we can use the type system to distinguish between the life and death of Anne and Tony. Listing 2.4 demonstrates a simple example of how AML will catch this error.

2.2.2 Calculation Bases and Their Types

Like products and risk models, AML calculation bases are defined in a state model. Listing 2.5 demonstrates a simple calculation basis. Note that the parameter `p` occurs both in the result type and in the body of

```
basis BasisLifeDeath(p : Person) : LifeDeath(p) where
  riskModel = RiskLifeDeath(p)
  interestRate = (t : TimePoint) => 0.05
  maxtime = p.birthDate + 120
```

Listing 2.5: Simple calculation basis

```
product SpouseBenefits(p : Person) : LifeDeath(p) where
  obligations = at t pay ¤1
                when(alive -> dead)
                provided(married)
                given(married ~ basis.marriageProb(p, t))
```

Listing 2.6: Survivor benefits

```
basis MarriageLifeDeath(p : Person) : LifeDeath(p) where
  riskModel = RiskLifeDeath(p)
  interestRate = (t : TimePoint) => 0.05
  maxtime = p.birthDate + 120

  marriageProb = (p : Person, t : TimePoint) => boolDist(0.4)
```

Listing 2.7: Basis for survivor benefits

the basis. The basis `BasisLifeDeath(anne)` could be used together with the product `TempLifeAnnuity(anne)` to calculate the reserves.

The product in Listing 2.6 introduces two new features of AML product definitions. The `given` construct binds a variable to a distribution in the product preceding it. In this case, `married` is the name of the bound variable. In the body of every product, a special variable `basis` is available. In order to match a product, a calculation basis must provide definitions for the fields that the product projects. For example, the calculation basis in Listing 2.7 defines the function `marriageProb` that `SpouseBenefits` references.

To support this, we expand product types to list the fields that they require and basis types to specify the extra fields that they provide, in a manner similar to record types. Then, a product type is compatible with a basis type if the basis has the necessary fields. Essentially, product and basis types include a record type. We expect to use ordinary record subtyping to represent the compatibility relationship, and we

speculate that banning or severely restricting dependent types in calculation bases will enable the record component of basis and product types to be completely inferred.

2.3 Continuing Work

Presently, the only implementation of AML is a dynamically-typed interpreter developed at Edlund A/S. Future work will include fully specifying the AML type system and implementing it, so that it can be tested with real users. We expect that AML will need to integrate with a number of external systems, including legacy pension management systems, customer databases, and foreign functions. This thesis' work on dependent type providers should be seen as an exploration of a potential feature of AML.

Part II

Background for Idris Type Providers

Chapter 3

F# Type Providers

This chapter describes a new feature called *type providers* that was introduced in version 3.0 of the F# programming language. A type provider is a compiler extension that is able to generate new types by executing arbitrary .NET code. Part III will demonstrate how a straightforward addition to the dependently-typed language Idris can achieve at least some of the benefits of type providers without the drawbacks inherent to code generation.

The purpose of type providers is to enable strongly-typed interaction with what Syme et al. [Sym+12] call "internet-scale information sources" — that is, very large datasets with fairly stable schemas that are not defined by or even necessarily under the control of a software developer who is interested in using them. Examples of type providers include an interface to the massive online database Freebase (containing 23,000 types), a type provider for CSV files, a client for Microsoft's OData protocol, and a regular expression library that can both statically check that a regular expression is well-formed and provide statically-checked accessors to portions of the input that were matched.

Traditional strategies for dealing with these kinds of situations in a statically-typed manner include:

1. Manually writing code to represent the schema
2. Automatically generating code to represent the schema
3. Neglecting to use the type system at all

None of these strategies are particularly satisfactory. The first strategy, manually representing the schema in the type system, provides a precise

```
type WorldBank =
  WorldBankDataProvider<"World Development Indicators">
let data = WorldBank.GetDataContext()

let usa = data.Countries.``United States``

printf "%A" usa.Indicators.``Population, total``
```

Listing 3.1: Using an F# type provider to print the USA's population

degree of control over the details of the representation. Unfortunately, it also *requires* a precise degree of control over the details of the representation. The labor burden associated with building such mappings and maintaining them in the face of changing schemas is likely to be significant. The second strategy, code generation, is likely to be more applicable to very large schemas. However, as Syme et al. [Sym+12] point out, code generation can be cumbersome, fitting especially poorly into workflows involving exploratory programming. The final option, an essentially dynamically-typed representation, fails to take advantage of all of the safety and tooling benefits of the statically-typed language.

Type providers address this situation by enabling flexible generation of types and values in the F# compiler. They can be seen as an extension of option (2) above. Because the types that are generated by type providers are full, first-class F# types, development tools have access to them. This means that environments such as Visual Studio can provide automatic completion of identifiers that are members of provided types and that provided types can have built-in documentation that will be shown in the development environment. Syme et al. [Sym+12] consider excellent tool support to be a key design goal and feature of F# type providers.

Type providers use an internal compiler API to create types as well as to generate program code. Type providers can take advantage of F# quotations [Sym06], which can simplify the generation of F# ASTs in some situations. The example type providers in Syme et al. [Sym+12] are written in an imperative style, where generated types are treated as mutable data structures that are updated with new members and attributes.

In order to support very large datasets and very large schemas, the type provider API provides the possibility for providers to be *lazy* and

```
type MyData = CsvProvider<"../files/example.csv">

let table = MyData.load("http://www.example.com/data.csv")
for row in table do
  printfn "%A %A" row.Col1 row.Col2
```

Listing 3.2: CSV type provider

the ability to *erase* their types. A lazy type provider does not actually generate its types until they are needed by the compiler. This is an important advantage over traditional code generation techniques when dealing with a very large schema. If a software developer only accesses some part of the data set, then types will only be generated for the portion that is used. Under type erasure, the provided types are removed from compiled code, being replaced by references to the first non-erased superclass of the erased class. This can have two benefits: it can reduce the size of compiled code, and it can make the compiled code more robust in the presence of minor schema changes.

As pointed out in Syme et al. [Sym+12], type providers introduce a number of potential pitfalls that are normally ruled out by the type system in a language such as F#. For example, type providers may generate types with cyclic inheritance graphs or references to other types that do not exist. They can even generate code that does not match the provided types. Because type providers operate "internally" to the language implementation, they bypass safety mechanisms that might prevent such errors in generated code.

Furthermore, type providers may rely on features of the world outside the program for their correctness. For example, run-time errors may occur if the schema of external data that the type provider relies upon has changed since the program was type checked.

As pointed out by Syme et al. [Sym+12], there is a certain surface similarity between F# type providers and dependent types. In fact, the syntax for providing value arguments to type providers, such as the filename of some CSV file, is just the ordinary F# syntax for type constructor application, as can be seen in the example in Listing 3.2. However, this similarity does not go deeper than the surface level.

A dependent type theory includes value-level computation as a part of the ordinary rules for comparing types while typechecking. In a dependently-typed language, new types can only be created from data to

the extent that the types can be parameterized or indexed by said data, and equivalence of types is restricted to equivalence of type constructors and of data. With a code generation paradigm, such as that of F#'s type providers, users are much more free to define new types. As such, many of the aforementioned problems with ill-formed types become possible, while a dependent type system would reject these types.

Syme et al. [Sym+12] exhibit a common confusion about dependent types when they state that "the types provided by an F# type provider do not depend on values computed at runtime" is a defining difference between dependent types and type providers, however. A distinction between compile-time and run-time computation exists in dependent type systems, as can be seen in Brady, McBride, and McKinna [BMM04] and in Edwin Brady's Ph.D. thesis [Bra05]. Dependent types merely enable the use of the same language to discuss compile-time and run-time computation rather than using separate languages for these purposes. In fact, Syme et al. hint at how to accomplish a similar task in a language with dependent types when they state that "pure dependent type systems can't base type-level computations on external information such as schemas" [Sym+12, p. 50]. As will be shown in Part III, simply extending dependent types with a mechanism for compile-time I/O, thereby somewhat reducing their purity, is sufficient to achieve at least some of the goals of type providers.

Chapter 4

Dependently Typed Programming

The goal of this chapter is to introduce just enough of the theory and practice of dependently-typed functional programming to enable readers who are already knowledgeable about Haskell or a dialect of ML to understand the remaining chapters of this thesis. As such, it will skip some topics that are traditionally part of introductions to dependently-typed programming, and it may include discussions that are not typically present in first introductions to dependent types. Additionally, certain details will be left out, such as completeness checking and tactic proofs. Readers who have knowledge of dependent types in advance can briefly skim this chapter to see how the constructs and idioms that they are already aware of are expressed in Idris. Please refer to the Idris Tutorial[1] for a more complete introduction.

While the term *dependent types* properly refers to any type system in which types can be parameterized [AH05], it is typically used to refer to systems in which types can quantify over *terms*, rather than systems such as ML in which they can only quantify over other *types*.

Perhaps the best-known examples of implemented programming languages with full dependent types are Agda [Agda] and Idris [Bra11]. There are also historically relevant languages such as Cayenne [Aug98] and Epigram [MM04; McB05]. Additionally, proof assistants such as Coq [Coq04] use dependent types as a logic, but are not really intended for dependently-typed *programming*. This chapter uses Idris's syntax in all examples.

[1] Available at the time of writing from `http://www.cs.st-andrews.ac.uk/~eb/writings/idris-tutorial.pdf`

```
data Nat : Type where
  O : Nat
  S : Nat -> Nat
```

Listing 4.1: The natural numbers

```
plus : Nat -> Nat -> Nat
plus O     m = m
plus (S n) m = S (plus n m)
```

Listing 4.2: Addition of natural numbers

```
data List : Type -> Type where
  Nil  : List a
  (::) : a -> List a -> List a
```

Listing 4.3: Lists

4.1 Example

The classic example to illustrate the utility of dependent types is the so-called *vector*, a kind of list whose type includes its length. This section presents the example, both to accustom readers to Idris's syntax and to provide a general introduction to dependent types in programming. Listing 4.1 defines a datatype `Nat` representing unary natural numbers. The first line states that `Nat` is a straightforward type with no indices or parameters. The second and third lines provide the means of constructing an instance of `Nat`. The constructor `O`, representing zero, takes no arguments and is already in `Nat`. The constructor `S`, representing the successor function, takes a `Nat` as an argument, returning another `Nat`. In this representation, three would be represented as `S (S (S O))`.

Addition of natural numbers can be defined by recursion on the first operand, as can be seen in Listing 4.2. In Idris as in Haskell, top-level pattern-matching definitions are written as a series of equations that map input patterns to output terms. The first equation whose left-hand side matches the arguments takes precedence over others that may match.

Listing 4.3 demonstrates the definition of linked lists. As in Haskell, a lower-case free variable (here `a`) receives implicit universal quantification. Behind the scenes, Idris inserts an implicit argument `a : Type`

```
map : (a -> b) -> List a -> List b
map f Nil       = Nil
map f (x :: xs) = f x :: map f xs

(++) : List a -> List a -> List a
(++) Nil       ys = ys
(++) (x :: xs) ys = x :: (xs ++ ys)
```

Listing 4.4: List operations

```
map' : (a -> b) -> List a -> List b
map' f Nil       = Nil
map' f (x :: xs) = map' f xs

(++!) : List a -> List a -> List a
(++!) Nil       ys = ys
(++!) (x :: xs) ys = (xs ++! ys)
```

Listing 4.5: Incorrect list operations

```
data Vect : Type -> Nat -> Type where
  Nil  : Vect a 0
  (::) : a -> Vect a n -> Vect a (S n)
```

Listing 4.6: Vectors

which the compiler is expected to find for the user.

Ordinary functions such as `map` and list concatenation (called `(++)`) are defined similarly to Haskell in Listing 4.4.

Again, the Haskell-style syntax for the type parameters `a` and `b` expands to implicit arguments to be filled out by the compiler. While Hindley-Milner type systems can prevent many errors in the above code, they cannot catch a missing element in the resulting list. Listing 4.5 demonstrates incorrect versions of `map` and `(++)`. Each of these ignore important aspects of their input. This is because the types are silent about the length of the list.

We can remedy this situation by using more specific types. First, we create a new inductive type `Vect`, which is a version of `List` that tracks its length in the type. Then, we add information about lengths to the types of our operations.

The `Nat` parameter to `Vect` is an ordinary instance of `Nat`. There is no complicated abuse of the type class mechanism *à la* McBride [McB02], circuitous encodings via implicit arguments and dependent method types *à la* Scala [OMO10] or limited extensions that only work with certain simple datatypes, such as the datakinds found in GHC Haskell [Yor+12].

A feature of dependent types is that the analog of the function type constructor `->` is a binding operator, just as λ-abstractions are in less expressive type theories. In other words, we can introduce new names to the left of an arrow that are then in the scope of the right of the arrow. We can see this clearly if we look at the version of the `Vect` constructor `(::)` where all the implicit arguments are explicitly enumerated:

```
(::) : {a : Type} -> {n : Nat} ->
       a -> Vect a n -> Vect a (S n)
```

The syntax `{n : Nat} -> ...` means that `n` is an implicit argument with type `Nat`. Furthermore, references to the name `n` in the remainder of the declaration refer to *the very same* `n` *that was passed*. With full dependent types, polymorphic types are merely functions that take types as arguments. Thus, we use the same mechanism to abstract over the type `a`.

We have defined a *family* of types that is *indexed* by the natural numbers — for each element n of `Nat`, there exists a type of vectors of precisely n elements. For every type `a`, the constructor `Nil` produces an instance of `Vec a 0`. That is, the empty vector contains zero elements. The constructor `(::)` takes an implicit natural number `n` as its parameter, which the compiler will be expected to find for users. Additionally, it expects a vector *whose length is precisely* `n`, and it produces a new vector whose length is precisely `S n`. Each instance of an identifier in a signature (for example `n` and `a` in the signature of `(::)`) refers to the same value.

Now, we can use `Vect` to define `map` and `(++)` such that an analog of the definitions in Listing 4.5 are ill-typed. In Listing 4.7, the type checker will reject the recursive calls in the `(::)` case because the resulting vector will be too short. More precisely, it will be unable to unify the index `n` from the tail with the index `S n` that is expected. The corrected versions can be seen in Listing 4.8.

Many tutorials on dependent types, having shown this convenient example, stop. In doing so, they neglect to illustrate an important con-

```
map' : (a -> b) -> Vect a n -> Vect b n
map' f Nil       = Nil
map' f (x :: xs) = map' f xs

(++!) : Vect a n -> Vect a m -> Vect a (plus n m)
(++!) Nil       ys = ys
(++!) (x :: xs) ys = (xs ++!  ys)
```

Listing 4.7: Incorrect (and ill-typed) vector operations.

```
map : (a -> b) -> Vect a n -> Vect b n
map f Nil       = Nil
map f (x :: xs) = f x :: map f xs

(++) : Vect a n -> Vect a m -> Vect a (plus n m)
(++) Nil       ys = ys
(++) (x :: xs) ys = x :: (xs ++ ys)
```

Listing 4.8: Correct (and well-typed) vector operations

```
plus' : Nat -> Nat -> Nat
plus' n O = n
plus' n (S m) = S (plus' n m)

(+++) : Vect a n -> Vect a m -> Vect a (plus' n m)
(+++) Nil       ys = ys
(+++) (x :: xs) ys = x ::  (xs +++ ys)
```

Listing 4.9: Alternate definition of addition and vector append

sequence that full dependent types can have for the way that we write programs. Imagine that `plus` was defined by recursion on its second argument. This definition represents the same mathematical function as the definition in Listing 4.2, in the sense that it will return the same result given the same arguments. Nevertheless, the definition of `(+++)` in Listing 4.9 will not typecheck. Here, we begin to see that full dependent types require a paradigm shift from users. In order to understand just why the new definition of `plus'` is more difficult to use with `Vect` and how to use it anyway, one must first understand a bit more about equality in dependent types.

`Vect a 0`		`Vect a m`		`Vect a (plus 0 m)`
`Nil`	`++`	`ys`	`=`	`ys`
`Vect a (S n')`		`Vect a m`		`Vect a (plus (S n') m)`
`x :: xs`	`++`	`ys`	`=`	`x :: (xs ++ ys)`

```
plus 0 m      ⟶  m
plus (S n') m ⟶  S (plus n' m)
```

Figure 4.1: Type checking vector concatenation

4.2 Definitional and Propositional Equality

Rather than taking a highly formal approach to the notions of equality that arise in type theory, this section examines them as they present themselves for users of a programming language, taking a highly operational approach. A more formal approach can be found in sources such as Aspinall and Hofmann [AH05]; Harper [Har12]; and Nordström, Petersson, and Smith [NPS90].

A typechecker must compare types for equivalence. Traditionally, this is done through unification after converting the type to some kind of normal form. This is also true in dependent types: the terms representing the type are normalized, and they are then checked for straightforward α-equivalence. This straightforward equality is known as *definitional equality*.

The original definition of `plus` worked conveniently in the type of `(++)` because the recursive structures of both functions were similar. Both functions are defined by recursion on the first argument, which has exactly one recursive occurrence of the type being eliminated. Figure 4.1 demonstrates the work that must be done to typecheck vector concatenation. First, examine the left side of each equation.

In the first equation, `Nil` must have type `Vect a 0`. We know that the type parameter is `a` because of the type declaration, and we know that the parameter `n` must be `0` based on the signature of the `Nil` constructor in Listing 4.6. The second argument, `ys`, provides no new information above that which is in the type signature, so its length is still `m`. Again from the type signature, we know that the right side must have type `Vect a (plus 0 m)`. However, we are provided with the term `ys`, which we know from the left side to have type `Vect a m`. In order for the first

```
data (=) : a -> b -> Type where
  refl : x = x
```

Listing 4.10: Propositional equality

case to typecheck, we must show that `Vect a (plus 0 m)` is equivalent to `Vect a m`. The term `plus 0 m` matches the first case of the pattern match in the original definition of `plus` in Listing 4.2, so we substitute the right-hand side and we are left with `m`, which gives us our solution.

A similar argument suffices to show that the second case is well-typed. We use the constructor that is being matched (namely `(::)`) to determine that the index `n` of the type of the first argument must be `S n'` for some `n' : Nat`. Then, we see that `plus (S n') m` matches the second equation for `plus` in Listing 4.2. Thus, it can be reduced to `S (plus n' m)`. Inductively from the type signature, we know that `xs ++ ys` will have type `Vect a (plus n' m)`. Then, applying the `(::)` constructor will yield the type `Vect a (S (plus n' m))`, which is what we were looking for.

This straightforward argument fails when `plus` is defined recursively on its second argument, as in `plus'` in Listing 4.9. This is because `plus'` always receives the variable `m` as its second argument in the definition of `(+++)`, so no reductions can be performed. In other words, `plus' 0 m` is no longer definitionally equal to `m` and `plus' (S n') m` is not definitionally equal to `S (plus' n' m)`.

In order to make `(+++)` typecheck with our redefined `plus`, we will need an additional tool. *Propositional equality* lifts the notion of two terms being equal into a datatype representing proofs of equality. While the equality datatype is a primitive in the high-level Idris language, it conceptually has the definition seen in Listing 4.10.

While the type constructor `(=)` can in principle accept any two types `a` and `b` as arguments and any of their elements, the constructor `refl` ensures that these elements in fact are the same element. This particular definition of propositional equality, where the types `a` and `b` are not necessarily the same, is sometimes referred to as "John Major equality" [McB99] or just heterogeneous equality.

An element of the type `x = y` constitutes a proof that `x` is in fact equal to `y`. This is because `refl` essentially copies its argument `x` to both indices of the type. While it is possible to use this evidence directly by defining the appropriate functions, Idris provides built-in syntax for

using equality proofs. The expression `rewrite P in tm`, where `P` has type `x=y`, replaces `x` with `y` in the *type* of `tm`, leaving the *term* `tm` intact.

The propositional equality type can be taken as our first example of the principle of *types as propositions*, or the *Curry-Howard Correspondence*. According to this principle, we can formulate *types* that represent propositions, and indexed type families that represent predicates, such that a member of a type serves as evidence for that proposition. Under this scheme, function types correspond to implication, dependent function types to universal quantification, and lambda terms to hypothetical assumptions. Similar constructions can be created for existential quantification, conjunction and disjunction, and so forth. While some writers distinguish between *mere propositions* that are singleton types whose proofs are therefore uninteresting and ordinary types that can have more than one element, this distinction is uninteresting for our purposes.

Now that we have a representation of equality as data and a means of using this equality to change types, we lack only one ingredient: the technique of actually constructing equality proofs. Because there are infinitely many elements of `Nat`, we cannot simply pattern-match all possible inputs. We can, however, write recursive functions, corresponding to proofs by induction.

4.3 Inductive Proofs

In order to make `(+++)` from Listing 4.9 typecheck, we must recover the salient aspects of the behavior of `plus` from the definition of `plus'`. We can do this by demonstrating two facts:

- for all `n : Nat, plus' O n = n`, corresponding to the first equation for `(+++)`, and
- for all `n : Nat` and `m : Nat`,
  `plus' (S n) m = S (plus' n m)`, corresponding to the second equation for `(+++)`.

Essentially, we must *prove* the facts that we got "for free" from the structure of our original definition of `plus`.

An informal proof that for all `n : Nat, plus' O n = n` follows. The proof proceeds by induction on `n`. Our base case is when `n = O`. We

```
plus_0_n_n : (n : Nat) -> plus' 0 n = n
plus_0_n_n 0      = refl
plus_0_n_n (S n') = let ih = plus_0_n_n n' in
                    rewrite ih in refl
```

Listing 4.11: Proof that for all `n : Nat`, `plus' 0 n = n`

must show that `plus' 0 0 = 0`. The left hand side reduces to `0`, and thus we must show `0 = 0`, which is true by reflexivity of equality. For the inductive step, we assume that `plus' 0 n = n`. We must show that `plus' 0 (S n) = S n`. Reducing according to the definition of `plus'` yields the goal `S (plus' 0 n) = S n`. By the induction hypothesis, we can replace `plus' 0 n` with `n`, yielding `S n = S n`, which is again true by reflexivity.

We can represent this proof in Idris as a recursive function. The case split corresponds to a pattern match. The base case is a defining equation with no recursive calls, while the inductive step corresponds to the equation containing a recursive call, with the induction hypothesis being the result of the recursive call.

Listing 4.11 demonstrates an Idris translation of the informal proof. The type signature simply states that `plus_0_n_n` is a function from any natural number `n` to evidence that `plus' 0 n = n`. This function is defined by a pattern match on the argument `n`, corresponding to the cases in the informal proof.

In our base case, we simply provide `refl` as evidence. The reduction of `plus' 0 0` that is explicit in the informal proof occurs implicitly during type checking, corresponding to the notion of definitional equality from Section 4.2.

In the induction step, we first bind the identifier `ih` to the result of the recursive call. Our induction hypothesis `ih` will then have the type `plus' 0 n' = n'`. Our goal, after replacing `n` with `S n'`, is a term with type `plus' 0 (S n') = (S n')`. The typechecker will reduce this goal to `S (plus' 0 n') = S n'`. We can use `refl` to construct a proof that `S n' = S n'`. This can be combined with the `rewrite` construct and the induction hypothesis to achieve the reduced goal — a proof that `S (plus 0 n') = S n'`.

Our other lemma is proved similarly, in Listing 4.12. Once again, we use induction on the second argument of `plus'`. In the induction step, we want to show that `plus' (S n) (S m) = S (plus' n (S m))`.

```
plus_Sn_m_Snm : (n, m : Nat) ->
                plus' (S n) m = S (plus' n m)
plus_Sn_m_Snm n 0 = refl
plus_Sn_m_Snm n (S m) =
  let ih = plus_Sn_m_Snm n m in
  rewrite ih in refl
```

Listing 4.12: Proof that for `n, m : Nat`, `plus' (S n) m = S (plus' n m)`

```
(+++) : Vect a n -> Vect a m -> Vect a (plus' n m)
(+++) {m=m} Nil ys = rewrite plus_0_n_n m
                     in ys
(+++) {n=S n} {m=m} (x :: xs) ys =
  rewrite plus_Sn_m_Snm n m
  in x :: (xs +++ ys)
```

Listing 4.13: Repaired definition of `(+++)`

Following reduction, we have

```
S (plus' (S n) m) = S (S (plus' n m))
```

Our induction hypothesis `ih` is evidence that `plus' (S n) m` is equal to `S (plus' n m)`. Thus, we can rewrite the goal to

```
S (S (plus' n m)) = S (S (plus' n m))
```

for which we can simply apply `refl`.

Now, we have shown that the features of `plus` that the typechecker relied on when checking `(++)` are also true for `plus'`. Listing 4.13 demonstrates how to use these facts to successfully typecheck `(+++)`. Because `n` and `m` are implicit arguments to `(+++)`, it is not normally possible to pattern-match on them or refer to them explicitly in the equations that define the function. The syntax `{ID=PAT}` matches the implicit argument `ID` to the pattern `PAT`. In each case, we use our lemmas about `plus'` to argue that the types are correct. The actual term remains unchanged from ordinary vector concatenation `(++)`.

A key advantage of dependently-typed programming is that the type system can verify expressive properties about our programs. As we have seen, structuring our definitions correctly makes this process much easier because the type theory uses the computational behavior of the

term language when checking for equality. If we cannot use this, then we need to write proofs by hand, which can be an involved process. We can use datatypes to represent proofs of many other properties than just equality. The first step, however, is to being writing our programs in a more information-rich style, or else we will never have the information that is necessary to prove anything.

4.4 Removing the Boolean Stench

The term *code smell*, popularized in Martin Fowler's book on refactoring [Fow99], refers to features of source code that are not incorrect *per se*, but may indicate a deeper problem. In a dependently-typed language, the use of `Bool` is widely considered to be a code smell. The problem is that Boolean values carry only one bit of information. They tell us that *something* is true or false, but are silent as to *what* was true or false. Thus, it is easy to accidentally use the wrong Boolean value. Even worse, once we have begun using Booleans, we are stuck. Because they are so information-poor, they prevent us from creating new information-rich data. For example, a Boolean resulting from an equality test cannot be used together with the `rewrite` construct the way that a `refl` can. This lack of information may necessitate creating more Booleans, as we no longer have the information that is necessary to construct further proof objects. In this way, Booleans spread through our programs like gangrene.

As an example of the kinds of information losses that can occur, imagine what would have happened in the previous section had we simply had a Boolean that resulted from comparing two numbers. We would not have been able to use the `rewrite ... in ...` construct. Because we had a type (namely `(=)`) whose element (namely `refl`) exploits the definitional equality of the type theory in order to carry around interesting information about the proof, we could get work done. By inventing new datatypes that capture information about our data, we can eliminate the Boolean stench.

4.4.1 List Membership

As an example of how to eliminate `Bool`-valued functions, we will convert the list membership function in Listing 4.14 into a procedure that

```
elem : Eq a => a -> List a -> Bool
elem x [] = False
elem x (y :: ys) = x == y || elem x ys
```

Listing 4.14: Boolean list membership

```
using (a : Type, x : a)
  data Elem : a -> List a -> Type where
    Here : Elem x (x :: xs)
    There : Elem x ys -> Elem x (y :: ys)
```

Listing 4.15: List membership as proposition

returns useful evidence as to whether or not `x` is a member of the list `xs`.

Sometimes, the Idris compiler is not sophisticated enough to infer the types of all implicit arguments. The `using` construct in Idris is simply a means of explicitly providing type annotations for implicit arguments across a number of definitions. In the indented block following `using (ID1 : TYPE1, ID2 : TYPE2, ...)`, the implicit variables $\texttt{ID}_n$ are assumed to have the corresponding types $\texttt{TYPE}_n$. Additionally, as Idris inserts implicit arguments from left to right, dependent types may not be in the correct order. A `using` block can be used to override the automatic ordering.

Listing 4.15 demonstrates a datatype whose members can serve as proofs of list membership. The constructor `Here` takes implicit arguments `x` and `xs` and is a proof that `x` is an element of the list `x :: xs`. That is, `x` is an element of any list in which it is the first element. The constructor `There` takes a proof that `x` is an element of `ys`, and returns a proof that `x` is an element of `y :: ys`, where `x`, `y` and `ys` are implicit arguments. In other words, if `x` is a member of the tail of a list, then it is a member of the list.

An instance of `Elem x xs` is more useful than a Boolean value for a number of reasons:

- it cannot be confused with an element of `Elem y ys`, while the result of `elem x xs` has the same type as a result of `elem y ys`
- it carries useful information with it — the number of `There` constructors on the result is the index at which the element can be found

```
data Dec : Type -> Type where
  Yes : p          -> Dec p
  No  : (p -> _|_) -> Dec p
```

Listing 4.16: Decidability

```
class DecEq a where
  decEq : (x, y : a) -> Dec (x = y)
```

Listing 4.17: Decidable equality

- a function that constructs an element of `Elem x xs` cannot return a "false positive" — the type checker ensures that the proof is really a proof

In order to fully replace our Boolean procedure, however, we require just a bit more machinery. We've seen how to represent that something is an element of a list, but we have not yet seen how to represent that it is not. Likewise, we do not yet have a counterpart to the Idris `Eq` typeclass and its `(==)` method. To demonstrate these, we use the notion of *decidable equality*.

4.4.2 Decidable Equality

So far, we have seen ways to construct elements of various types as the evidence for the truth of some proposition represented by the type. We have not, however, seen how to represent the negation of such a proposition. In Idris, there is a type called `_|_` (meant to look like the mathematical symbol $\bot$) which has no constructors. In other words, it is impossible to construct a term whose type is `_|_`. Thus, by the propositions-as-types principle, `_|_` can serve as the false proposition, as it can never be proven.

We can thus represent the negation of a proposition `P` using the type `P -> _|_`. We should expect that our counterpart to `elem` will return either a proof `Elem x xs` or a proof `Elem x xs -> _|_`. These options are captured in the `Dec` datatype in Listing 4.16.

Note that the law of the excluded middle in classical logic implies that every proposition is either true or false. Thus, we should be able to create an instance of `Dec p` for *every* type `p`. However, in constructive logics such as type theory, we have no such law. We call a function

```
pmEx : (x : a) -> (y : a) -> Maybe (x = y) -> ()
pmEx x x (Just refl) = ()
pmEx x y Nothing     = ()
```

Listing 4.18: Matching the equality type

returning `Dec p` for some type `p` a *decision procedure*. In the presence of a decision procedure, we in some sense locally recover the law of the excluded middle.

In particular, we may be interested in whether equality or inequality can be determined for any two members of some type. The type class `DecEq a` (demonstrated in Listing 4.17) represents the fact that equality is in fact decidable for type `a`.

4.4.3 Dependent Pattern Matching

Sometimes, pattern-matching one argument in the presence of dependent types will reveal information about the form of other arguments. This can introduce *non-linear patterns* and *expressions as patterns*. Pattern matching in dependent types was introduced by Coquand [Coq92], and an accessible presentation of its potential can be found in the papers on Epigram [MM04; McB05].

For example, inspecting an argument of type `x = y` (that is, matching against the pattern `refl`) introduces new information about `x` and `y` — namely, that they are equal. As such, either occurrences of `x` must be replaced by `y` or occurrences of `y` must be replaced by `x` in the rest of the pattern.

Listing 4.18 demonstrates this mechanism in action. Because the third argument in the first equation contains a constructor for `x = y`, we *must* repeat the same name for both instances of `x`. In the second equation, we *must not* repeat the pattern variable, as we have no reason to believe that they should be equal. Just as we can find out that two pattern variables in fact are the same by pattern matching an equality, we can also find out that a pattern must be equal to some expression. Listing 4.19 demonstrates a simple example of how this works. Because our pattern `refl` forces the two expressions in its type to be equal, we know that the first argument to `pmEx2` must be `plus (S 0) (S 0)`, or $1 + 1$.

```
pmEx2 : (x : Nat) -> x = plus (S O) (S O) -> ()
pmEx2 (plus (S O) (S O)) refl = ()
```

Listing 4.19: Expressions in patterns

```
data Cmp : Nat -> Nat -> Type where
  Lt : (n, d : Nat) -> Cmp n (n + S d)
  Eq : (n : Nat) -> Cmp n n
  Gt : (d, n : Nat) -> Cmp (n + S d) n
```

Listing 4.20: Comparison results

Idris also implements the `with` rule, a generalization of Haskell's pattern guards [EPJ01] that was first described by McBride and McKinna [MM04]. The `with` rule allows dependent pattern matching of additional expressions in the same context as the arguments to the pattern match. That is, equalities introduced by the type of the additional expression are manifested as nonlinear patterns just as if they were a part of the original set of values being matched against.

We can deodorize the ordinary comparison functions in a manner demonstrated by McBride and McKinna [MM04]. Listing 4.20 contains a datatype `Cmp`. The two `Nat` indices are used to witness the following observation: if $a > b$ then there exists some δ such that $a = b + \delta + 1$. In the constructors `Lt` and `Gt` in Listing 4.20, `d` represents the difference δ and `n` represents the smaller number. McBride and McKinna observe that comparing two `Nats` requires subtracting them anyway, so keeping the result of this subtraction around (that is, δ) may additionally avoid repeated work and simplify definitions that rely on the difference. Additionally, the indices on `Cmp` make it impossible to return `Gt` when `Lt` would have been the correct answer.

Listing 4.21 demonstrates a function that, given a pair of natural numbers, generates a corresponding `Cmp` value. The first three cases are straightforward, resting on the observations that $0 = 0$ and $n - 0 = n$. The fourth case employes the `with` rule. It recursively calls `cmp` on the predecessors of its arguments, making the result available for inspection by pattern matching. The result of the recursive call to `cmp` is available for pattern-matching to the right of the vertical bar character in the following indented code block. Just as in Listing 4.19, the constructors of `Cmp` induce equalities between pattern variables and expressions, which

```
cmp : (n, m : Nat) -> Cmp n m
cmp 0     0     = Eq 0
cmp 0     (S n) = Lt 0 n
cmp (S n) 0     = Gt n 0
cmp (S n) (S m) with (cmp n m)
  cmp (S n) (S (n + S d)) | Lt n d = Lt (S n) d
  cmp (S n) (S n)         | Eq n   = Eq (S n)
  cmp (S (m + S d)) (S m) | Gt d m = Gt d (S m)
```

Listing 4.21: Comparing Nats

must be reflected in the corresponding cases to the left of the vertical bar. As such, we replace either `m` or `n` with the appropriate sum in the `Lt` and `Gt` sub-cases and we repeat the variable in the `Eq` case. Then, the comparison result for `S n` or `S m` is created, maintaining the same difference `d`.

This is a common theme in dependently-typed programming: creating informative datatypes that ensure that programs are correct by construction. We now turn to using these principles to replace the Boolean list membership function from Listing 4.14.

4.4.4 List Membership

Now that we have seen decision procedures for propositional equality (that is, the `DecEq` typeclass) and how to use equality proofs while pattern-matching, we can construct a `Bool`-free counterpart to `elem` from Listing 4.14. Listing 4.22 contains a the basic outline of our function. The return type, `Dec (Elem x xs)`, will either be a proof that `x` is in `xs` or a proof that it is not.

The function must be defined by pattern-matching on its second argument, the list, because we know nothing about the structure of `x`. Difficulties present themselves already in the first case, the empty list. We know that `x` cannot occur in the empty list, so we will need the `No` constructor of `Dec`, but how do we *prove* that `x` is not in `[]`?

To show that a function returns the empty type, one must simply demonstrate that there are no type-correct ways of calling it. In Idris, this is typically accomplished by specifying enough constructors in a pattern-match to induce the type checker to fail unification somewhere. Instead of a right-hand side, these cases are decorated with the

```
decElem : (x : a) -> (xs : List a) -> Dec (Elem x xs)
decElem x []        = No ???
decElem x (y :: ys) = ???
```

Listing 4.22: Outline of list membership

```
using (a : Type, x : a)
  notNilElem : Elem x [] -> _|_
  notNilElem Here      impossible
  notNilElem (There _) impossible
```

Listing 4.23: The empty list cannot have members

```
decElem : (x : a) -> (xs : List a) -> Dec (Elem x xs)
decElem x []        = No notNilElem
decElem x (y :: ys) = ???
```

Listing 4.24: List membership: nothing can be in the empty list

`impossible` keyword.

In Listing 4.23, the first match is impossible because `Here`'s second index must unify with the `(::)` constructor. In the type signature, however, the `Nil` constructor is provided. Likewise, the constructor `There` also uses a `(::)` in its second index. This case must also be impossible. Thus, we know that nothing can be an element of the empty list.

We can use this proof as the argument to `No` in our first equation for `decElem`, yielding the code in Listing 4.24. This corresponds to the base case in our Boolean function `elem` — we are simply returning something more informative than a mere `False`.

In order to determine whether `x` is an element of `y :: ys`, we must determine two things: whether `x` and `y` are equal, and whether `x` is an element of the tail `ys`. Again, this parallels the structure of the second case of our Boolean `elem`.

We have already seen how to determine whether `x` and `y` are the same: the `decEq` method from the `DecEq` typeclass. If they are the same, then we can use the `with` rule to witness this fact, and then the `Here` constructor. In Listing 4.25, `with` has been used to apply our equality decision procedure to `x` and `y`.

If `x` is not equal to `y`, then we need to check whether `x` occurs in `ys`. We do this through a recursive call to `decElem` at the site of our `with`

```
decElem : DecEq a =>
          (x : a) -> (xs : List a) -> Dec (Elem x xs)
decElem x []        = No notNilElem
decElem x (y :: ys)   with (decEq x y)
  decElem x (x :: ys) | Yes refl = Yes Here
  decElem x (y :: ys) | No notHere = ???
```

Listing 4.25: List membership: check the head

```
decElem : DecEq a =>
          (x : a) -> (xs : List a) -> Dec (Elem x xs)
decElem x []        = No notNilElem
decElem x (y :: ys)   with (decEq x y, decElem x xs)
  decElem x (x :: ys) | (Yes refl, _) = Yes Here
  decElem x (y :: ys) | (_, Yes inRest) =
    Yes (There inRest)
  decElem x (y :: ys) | (No notHere, No notThere) =
    No ???
```

Listing 4.26: List membership: check the tail

rule. If x in fact is a member of ys, then the recursive call will yield Yes applied to a proof of that fact. We use this proof with There to construct a proof that x is an element of `y :: ys`, and wrap it in the Yes constructor.

If we know that x is neither the head of nor in the tail of xs, then we know that it is not an element of xs. In order to return No, we must construct a witness of this fact. Examining our pattern match, we have terms `notHere : x = y -> _|_` and `notThere : Elem x ys -> _|_`. We must use this to construct a proof that `Elem x (y :: ys) -> _|_`. In other words, we will need another auxiliary lemma with type:

```
(x = y -> _|_) ->
(Elem x ys -> _|_) ->
Elem x (y :: ys) -> _|_
```

Unlike notNilElem, we cannot achieve this through pattern-matching and pointing out to the type checker that the employed constructors cannot be unified. Our argument of type `Elem x (y :: ys)` is the only one that is an inductive datatype. Thus, we perform a case analysis, yield-

```
using (a : Type, x : a)
  notConsElem : (x = y -> _|_) ->
                (Elem x ys -> _|_) ->
                Elem x (y :: ys) -> _|_
  notConsElem notHere notThere Here = notHere refl
  notConsElem notHere notThere (There r) = notThere r
```

Listing 4.27: Cons cell membership must be head or tail

ing equations for `Here` and `There`. Each of these cases allows us to use one of the other arguments to derive `_|_`. This definition can be seen in Listing 4.27. Each right hand side uses the argument corresponding to its constructor together with the evidence introduced from the pattern match. The function could easily be polymorphic, returning some type `a` instead of `_|_`, but the specialized version is shown here for the sake of simplicity.

Finally, we can combine `notConsElem` with our two `No` results in Listing 4.26 to construct our final proof of non-membership. Recall that the `No` constructor accepts an argument of type `p -> _|_`. This is a function type. In our particular case, `p` is `Elem x (y :: ys)`. Thus, `No`'s argument can be a λ-abstraction. In the body of the abstraction, we use `notConsElem` to show `_|_`.

The completed result can be seen in Listing 4.28.

4.5 Universe Construction

The final tool that will be necessary to understand the presentation of Idris type providers is the common pattern in dependently-typed programming of defining *universes*. Universes function as a kind of "design pattern" in the sense of Gamma, Helm, Johnson, and Vlissides [Gam+94]. As with the other features, this section will only explain this deep topic to the extent that is necessary to understand Idris type providers.

4.5.1 Motivating Example

Heterogeneous collections in the spirit of Kiselyov, Lämmel, and Schupke [KLS04] are straightforward in Idris. For example, a hetero-

```
using (a : Type, x : a)
  notNilElem : Elem x [] -> _|_
  notNilElem Here      impossible
  notNilElem (There _) impossible

  notConsElem : (x = y -> _|_) ->
                (Elem x ys -> _|_) ->
                Elem x (y :: ys) -> _|_
  notConsElem notHere notThere Here = notHere refl
  notConsElem notHere notThere (There r) = notThere r

decElem : DecEq a =>
          (x : a) -> (xs : List a) -> Dec (Elem x xs)
decElem x [] = No notNilElem
decElem x (y :: ys)   with (decEq x y, decElem x ys)
  decElem x (x :: ys) | (Yes refl,   _ ) = Yes Here
  decElem x (y :: ys) | (_, Yes inRest ) =
    Yes (There inRest)
  decElem x (y :: ys) | (No notHere, No notThere) =
    No (\h => notConsElem notHere notThere h)
```

Listing 4.28: Complete code for Boolean-free list membership

```
data HList : List Type -> Type where
  Nil : HList []
  (::) : (x : t) -> HList ts -> HList (t :: ts)

foo : HList [Nat, String, Nat]
foo = [42, "fnord", 13]
```

Listing 4.29: Heterogeneous lists in Idris

geneous list need only be indexed over a list of types. Listing 4.29 demonstrates such a definition.

However, this kind of `HList` affords too much freedom for certain purposes. In essence, it is a nested tuple, and we can use ordinary polymorphism to write functions that *e.g.* extract the first element of a non-empty `HList`, such as `fst` in Listing 4.30. We cannot, however, easily write functions that work with `HLists` in general, such as `map` or `foldl`. This is because the function being mapped would need to accept

```
fst : HList (t :: ts) -> t
fst (x :: xs) = x

test : Nat
test = fst foo
```

Listing 4.30: Extracting the first element

input of any type, and could therefore be only a constant function or the identity function.

4.5.2 Mapping across a universe

To define a useful `map` function for `HList`, we need a means of knowing something more about the types that are contained in the HList. There are a number of ways to do this:

- In a language such as recent versions of GHC Haskell with typeclasses and constraint polymorphism, we could require that all types in the `HList` have an available typeclass instance. The function being mapped could then make use of the methods from the typeclass.
- In a language with subtyping and bounded polymorphism such as Scala [Ode+04], we could require that all types in the `HList` have some common bound, which the function being mapped could exploit.
- Traditional existential types could be used to package a type together with enough information to perform the function.

Idris does not have constraint polymorphism or subtyping, and while existentials can be modeled, they are not particularly convenient. Another option is to limit the types contained in the list to some restricted, finite collection of types — a particular *universe* of types.

This can be achieved straightforwardly. We first define a datatype representing the structure of the particular types that we wish to include in our universe. Elements of this datatype are referred to as *codes* for the types that they represent. Andjelkovic's M.Sc. thesis [And11] goes into great detail about the varieties of generic programming that are available with different kinds of universes — for the purpose of this example, we

```
data U = NAT | STRING | LIST U

interp : U -> Type
interp NAT = Nat
interp STRING = String
interp (LIST u) = List (interp u)
```

Listing 4.31: A simple universe

```
data UMaybe : U -> Type where
  UNothing : UMaybe t
  UJust : (t : U) -> interp t -> UMaybe t
```

Listing 4.32: A Maybe for our universe

```
data UMaybe : (u : Type) -> (u -> Type) -> u -> Type where
    UNothing : UMaybe u el t
    UJust : (t : u) -> el t -> UMaybe u el t
```

Listing 4.33: Polymorphic `Maybe` type for a universe

will use a very simple universe consisting of natural numbers, strings, and lists of other types in the universe. Listing 4.31 demonstrates the datatype `U` and a mapping `interp` from the datatype to types. Taken together, these two elements define our universe.

A simple example of what we can do with a universe can be seen in Listing 4.32. Instead of using a type parameter to represent the type of object we should expect to find in a `Just` constructor, we use an element of `U`. Then, in `UJust`, we use `interp` to determine what type of element we in fact should store. Later, functions can pattern-match on the first parameter of `UJust` to find out what type of data is stored.

In fact, we can generalize `UMaybe` to work with *any* universe. As a convention, `t` will be used as a variable that quantifies over particular instances of universe datatypes, `u` will be used to quantify over these datatypes themselves, and `el` will quantify over the interpretation function. Listing 4.33 strips out the reliance on `U` and `interp` from `UMaybe`.

Now, we are prepared to define `map` for a universe-bounded `HList`. Listing 4.34 demonstrates heterogeneous lists in a universe. Like `UMaybe`, `UList` has a universe type `u` and a mapping from `u` to `Type` as parameters. Additionally, `UList` is indexed by `List u`, similarly to the way that

```
data UList : (u : Type) -> (u -> Type) ->
             List u -> Type where
  Nil : UList u el []
  Cons : (t : u) -> el t ->
         UList u el ts ->
         UList u el (t::ts)

(::) : {t : u} -> el t -> UList u el ts -> UList u el (t::ts)
(::) {t=t} = Cons t

foobar : UList U interp [NAT, STRING, NAT]
foobar = [42, "fnord", 13]
```

Listing 4.34: Heterogeneous lists in a universe

```
size : (t : U) -> interp t -> Nat
size NAT      n         = n
size STRING   s         = length s
size (LIST t) []        = 0
size (LIST t) (x :: xs) = 1 + size t x + size (LIST t) xs
```

Listing 4.35: The size of values from our universe

`HList` is indexed by a list of codes from our universe. `Cons` is defined separately from the `(::)` operator because it is easier to pattern-match when `t` is explicit, but it is easier to construct lists when it is implicit.

Returning to our universe `U`, we can define a notion of size for values drawn from this universe. The size of a natural number is merely the number itself, the size of a string is its length, the size of the empty list is 0, and the size of a cons cell is the one plus the size of the head plus the size of the tail. This operation is defined in Listing 4.35. Note that the value of `t` found in the first argument that we pattern-match on determines the type of the second argument, and thus the range of patterns that make sense.

The function `size` has a structure typical to those that we would like to map across our `UList`. These functions should take two arguments: a code in `U` and an element of the type corresponding to that code. Then, `map` looks quite similar to the ordinary definition of `map` for `List`. It simply provides the function to be mapped with code along with the value. A more advanced implementation could accept a new

```
map : (f : (t : u) -> el t -> r) -> UList u el ts -> List r
map f Nil           = []
map f (Cons t x xs) = (f t x :: map f xs)

bigness : List Nat
bigness = map size foobar

-- From the REPL:
-- > bigness
-- [42,5,13] : List Nat
```

Listing 4.36: Mapping in our universe

universe `u'` and `el'`, a universe mapping `m` in `u -> u'`, and then map a function in `(t : u) -> el t -> el' (m t)`. This would then return a `UList u' e' (map m ts)` but such genericity would complicate the example needlessly.

4.6 Toward Dependent Type Providers

Type providers for Idris simply combine the tools that we have seen with the ability to execute I/O actions at compile-time. A type provider will then typically consist of a universe describing the provided types as well as a means of reading the codes for the types from the environment. This can be used together with a suitable library that uses these universes to give features similar to type providers in F#. Part III is dedicated to demonstrating just how this works.

Part III

The Technique

Chapter 5

Idris Type Providers

While F# type providers are a substantial technical achievement, there is also a downside. In some sense, they break the internal consistency of the language. Instead of using the native abstraction capabilities of F#'s type system, type providers use code generation to create F# ASTs. This is necessary, because type providers inherently convert data (such as schema descriptions) into types, and since F# does not have dependent types, some form of code generation is the only way to do this. The question then arises: in a language *with* full dependent types, is it possible to solve the problems that are solved by F#'s type providers without resorting to code generation?

This chapter presents our technique for allowing Idris types to depend on the results of arbitrary I/O actions, performed while typechecking. This is the first major contribution of this thesis.

5.1 Example

A very simple type provider is one that reads a file and chooses a type based on the contents of the file. Listing 5.1 contains an Idris program that will, during typechecking, read the contents of the file `"theType"`. If the file contains the string `"Int"`, then `T1` will be bound to the type `Int`. If the file contains any other string, then `T1` will be bound to `Nat` instead. Because `2` is valid syntax for either `Int` or `Nat` values, the module will be syntactically correct in either case.

Note that while `%provide (T1 : Type)` would seem to be specifying that `T1` should be a `Type`, `fromFile` is a function from `String` to `IO (Provider Type)`. As in Haskell, the `IO` type constructor represents

```
module Main

import Providers

%language TypeProviders

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

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

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

foo : T1
foo = 2
```

Listing 5.1: A simple Idris type provider, as they are described in this chapter

a monadic language for constructing programs that can have side effects. The `Provider` type constructor and the `Provide` data constructor are a component of the type providers library. For now, it suffices to know that `Provide` has the signature `a -> Provider a`.

While typechecking, the Idris compiler will execute the program `fromFile "theType"`, performing any necessary side effects. The result is then unpacked from the `Provider` constructor and bound to `T1` for the remainder of typechecking.

5.2 Error Handling

An important practical feature of a static type system is the ability to provide informative error messages to users. However, most of the interesting activity in type providers occurs "outside" the type system, during execution. Thus, we cannot rely on Idris's type checker to return informative error messages. This is why type providers are not simple terms in `IO a`. Instead, they are in `IO (Provider a)`.

The `Provider` datatype, shown in Listing 5.2, has two constructors.

```
data Provider a = Provide a | Error String
```

Listing 5.2: The Provider datatype

As we have seen, `Provide` is a simple wrapper around a provided result, while `Error` is a means for type providers to fail with a useful error message. `Provider a` is isomorphic to the more usual `Either String a` — it is a separate type merely out of a pragmatic wish to make it obvious when something as risky as compile-time I/O will occur.

Listing 5.3 illustrates a simple, silly provider that takes advantage of this error-handling mechanism. Additionally, it 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 again until it succeeds. When an integer is successfully parsed, `adultsOnly` then checks whether it is at least `18`. If the user is over 18 years old, it provides the Boolean value `True`, and if the user is not yet 18, it refuses to typecheck. Obviously, this is easy for underage programmers to defeat — nevertheless, it is an effective demonstration of the error mechanism.

Omitted from Listing 5.3 are the straightforward functions `parseInt` with type `String -> Maybe Int` and the standard library function `trim`, which removes leading and trailing whitespace from a string.

5.3 Type Providers vs. Data Providers

Because dependently-typed programming languages do not have a fundamental distinction between the terms that represent values or computations and the terms that represent types, a type provider mechanism is necessarily also a data provider mechanism. While the term “type provider” is used to discuss the feature, keep in mind that it will often be ordinary data that is returned.

Note that this is also the case for F# type providers. The types that they generate are within the .NET system, which means that they are classes with members. Clearly, there must also be a provided means of constructing these objects, so F# type providers also provide data.

The indexed datatypes discussed in Chapter 4 provide another motivation for returning data from type providers. If we are to return

```
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
```

Listing 5.3: A type provider for adults

type-valued expressions, we must necessarily be able to construct the parameters to these types. In fact, in many cases, we may be able to get away with only generating indices to some datatype that exists completely external to the type provider mechanism.

5.4 Type Provider Semantics

This section presents the semantics of type providers in Idris. Idris type providers are simply expressions of type `IO (Provider a)` for some type `a`. They do not have access to the internals of the compiler, and there is no special API available to them beyond the ordinary API of the `IO` monad. This has an important consequence: because they are not based on code generation, they can be used in the module in which they were defined. Additionally, it means that it is sufficient to define the semantics of the `%provide` declaration in terms of Idris's semantics in order to provide a description of the semantics of type providers.

5.4.1 Elaborating Idris

Idris, like many other systems, has a tiny core language with a trusted typechecker. The features of the high-level languages are first translated to this core language, in a process called *elaboration*, after which the core language is typechecked and possibly compiled. (Note that this is different from usage in the ML community [Mil+97], where "elaboration" refers to the entire application of the language's static semantics.) Idris's core type theory, called TT, is described by Brady [Bra13]. It is a very simple dependent type theory, with a notion of pattern-matching top-level definitions.

Briefly, the elaboration process makes all implicit arguments explicit. It replaces type class constraints with ordinary parameters and type class method invocations with explicit references to the dictionary parameters from the constraint. Idris's `with`-clauses, `where`-clauses, and Haskell-style `case` expressions are lifted to top-level definitions. Elaboration occurs by means of a sort of embedded monadic tactic language in the Haskell implementation.

If at all possible, new features of Idris should be defined in terms of their elaboration to TT definitions and terms, rather than by extending TT. This allows us to maintain our trust in the carefully-checked implementation of the core theory.

Edwin Brady's paper on Idris [Bra13] provides a complete definition of TT. The present thesis follows the conventions of that paper in using `typewriter font` to represent high-level Idris code and conventional mathematical notation to represent TT terms. Some of Idris's high-level features, such as `do`-notation and idiom brackets, are defined by a straightforward syntactic transformation on high-level Idris. Others, namely `case` expressions and metavariables, build upon the elaboration of the other features of the language but can also define new top-level definitions as a side effect of elaboration.

5.4.2 Elaborating Type Providers

In order to define the elaboration of Idris type providers, we first need a few tools. In this section, C refers to a (possibly empty) global context of TT declarations and definitions, while Γ refers to a (possibly empty) local context consisting of a sequence of binders $\lambda x : t$, $\forall x : t$, or $\underline{\text{let}}\ x : t \mapsto t'$. An empty global or local context is written as a dot "$\cdot$".

$$C;\Gamma \vdash t \xrightarrow{\text{exec}} t' \ : \ \tau \qquad\qquad C;\Gamma \vdash t \ \underline{\text{done}}$$

$$C;\Gamma \vdash t_I \xrightarrow{\text{elabT}} \left(t_{\mathsf{TT}} \ ; \ \Gamma' \ ; \ \overrightarrow{d_{\mathsf{TT}}}\right) \qquad C;\Gamma \vdash t_I \xrightarrow{\text{elabT}}\!\!\circ \ err$$

$$C \vdash d_I \xrightarrow{\text{elabD}} \overrightarrow{d_{\mathsf{TT}}} \qquad\qquad C \vdash d_I \xrightarrow{\text{elabD}}\!\!\circ \ err$$

Figure 5.1: Judgments

$$\begin{array}{l}\underline{\texttt{data}} \quad \texttt{IO}\ (a\ :\ \texttt{Type})\ :\ \texttt{Type} \quad \underline{\texttt{where}} \\ \qquad\quad \texttt{IOCon}\ :\ (x\ :\ a) \to \texttt{IO}\ a\end{array}$$

$$\begin{array}{l}\underline{\texttt{data}} \quad \texttt{Provider}\ (a\ :\ \texttt{Type})\ :\ \texttt{Type} \quad \underline{\texttt{where}} \\ \qquad\quad \texttt{Provide}\ :\ (x\ :\ a) \to \texttt{Provider}\ a \\ \qquad \mid\ \texttt{Error}\ :\ (err\ :\ \texttt{String}) \to \texttt{Provider}\ a\end{array}$$

Figure 5.2: TT datatypes used by type providers

The context will make clear which is which.

Next, we need some notion of *execution*, which is intended to be equivalent to a sequence of actions that the language runtime will perform when executing a program in the `IO` monad. Since the specific details of the execution semantics of each construct in TT would be tedious, requiring a detailed specification of features such the C FFI, we simply define the relevant properties of the execution relation. Any execution semantics that has these properties could be used as a basis for type providers. The judgment $C;\Gamma \vdash a \xrightarrow{\text{exec}} b$ indicates that the result of executing a is b, in the global context C and the local context Γ. The judgment $\Gamma \vdash e \ \underline{\text{done}}$ indicates that execution of e has reached a final value form — that is, that its execution did not get "stuck". For the execution of terms, we assume preservation but not necessarily progress. That is, we allow terms being executed to get "stuck", but we do not allow them to change types. Thus, the execution relation is not typesafe in the traditional sense.

We take `IO` and `Provider` to be the TT types declared in Figure 5.2. The latter is the result of elaborating the definition in Listing 5.2. In the

$$\textsf{ExecPres}\ \frac{C;\Gamma \vdash a\ :\ \texttt{IO}\ T \quad C;\Gamma \vdash a \xrightarrow{\textsf{exec}} b}{C;\Gamma \vdash b\ :\ \texttt{IO}\ T}$$

$$\textsf{ExecDone}\ \frac{}{\Gamma \vdash \texttt{IOCon}\ x\ \underline{\text{done}}}$$

Figure 5.3: TT execution properties

actual implementation, `Provider` is an ordinary Idris datatype, while `IO` is treated specially by the compiler. In particular, `IOCon` has no user-accessible name in Idris.

Brady [Bra13] is concerned with the nitty-gritty details of transforming the highly implicit Idris to the completely explicit TT through an embedded tactic proof language in Haskell. As such, the paper has a highly operational approach, demonstrating monadic tactic scripts to elaborate each feature. However, elaboration of type providers is quite straightforward, while explaining the semantics of Brady's tactic language would be quite involved. Thus, the process is instead presented here using inference rules. The typing relation has the form $C;\Gamma \vdash t_1\ :\ t_2$. This represents that t_1 can have type t_2 in the global context C and the local context Γ. Terms at the top level are typed in the empty local context, but they have access to earlier 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 $C;\Gamma \vdash t_I \xrightarrow{\textsf{elabT}} \left(t_{\textsf{TT}}\ ;\ \Gamma'\ ;\ \overrightarrow{d_{\textsf{TT}}}\right)$ states that in the global context C and the local context Γ, the Idris term t_I elaborates to the TT term $t_{\textsf{TT}}$, producing a new local context Γ' and the possibly-empty sequence of new top-level definitions $\overrightarrow{d_{\textsf{TT}}}$. The corresponding error judgment $C;\Gamma \vdash t_I \xrightarrow{\textsf{elabT}}\!\!\circ\ \mathit{err}$ indicates that elaboration of t_I fails with some error err.

The judgment $C \vdash d_I \xrightarrow{\textsf{elabD}} \overrightarrow{d_{\textsf{TT}}}$ indicates that the Idris top-level definition d_I is successfully elaborated to the sequence of TT definitions $\overrightarrow{d_{\textsf{TT}}}$. The corresponding error judgment $C;\Gamma \vdash d_I \xrightarrow{\textsf{elabD}}\!\!\circ\ \mathit{err}$ indicates that elaboration of the Idris definition d_I fails with some error err.

Figure 5.4 describes the process of successful type provider elaboration. First, the declared type is elaborated to some TT term τ. Assuming

$$\frac{\begin{array}{rr} C;\cdot \vdash T \xrightarrow{\text{elabT}} \left(\tau \;;\; \Gamma_1 \;;\; \overrightarrow{d_1}\right) & C, \overrightarrow{d_1};\cdot \vdash \tau \;:\; \texttt{Type} \\ C, \overrightarrow{d_1};\cdot \vdash p \xrightarrow{\text{elabT}} \left(\pi \;;\; \Gamma_2 \;;\; \overrightarrow{d_2}\right) & C, \overrightarrow{d_1}, \overrightarrow{d_2};\cdot \vdash \pi \;:\; \texttt{IO}\ (\texttt{Provider}\ \tau) \\ C, \overrightarrow{d_1}, \overrightarrow{d_2};\cdot \vdash \pi \xrightarrow{\text{exec}} \texttt{IOCon}\ (\texttt{Provide}\ t) & \end{array}}{C \vdash \%\texttt{provide}(x : T)\ \texttt{with}\ p \xrightarrow{\text{elabD}} \overrightarrow{d_1}\ ,\ \overrightarrow{d_2}\ ,\ x \;:\; \tau\ ,\ x = t}$$

Figure 5.4: Successful elaboration of type providers

$$\frac{\begin{array}{rr} C;\cdot \vdash T \xrightarrow{\text{elabT}} \left(\tau \;;\; \Gamma_1 \;;\; \overrightarrow{d_1}\right) & C, \overrightarrow{d_1};\cdot \vdash \tau \;:\; \texttt{Type} \\ C, \overrightarrow{d_1};\cdot \vdash p \xrightarrow{\text{elabT}} \left(\pi \;;\; \Gamma_2 \;;\; \overrightarrow{d_2}\right) & C, \overrightarrow{d_1}, \overrightarrow{d_2};\cdot \vdash \pi \;:\; \texttt{IO}\ (\texttt{Provider}\ \tau) \\ C, \overrightarrow{d_1}, \overrightarrow{d_2};\cdot \vdash \pi \xrightarrow{\text{exec}} \texttt{IOCon}\ (\texttt{Error}\ \mathit{err}) & \end{array}}{C \vdash \%\texttt{provide}(x : T)\ \texttt{with}\ p \xrightarrow{\text{elabD}} \circ\ \mathit{err}}$$

Figure 5.5: Failed elaboration of type providers

that τ is in fact a type, the provider term is then elaborated. If the elaborated provider term has the appropriate type `IO (Provider` τ`)`, then it is executed, and if execution terminates normally, the resulting term is inspected. If the resulting term was `Provide` t, then the entire `%provide` clause is elaborated to the declaration $x \;:\; \tau$ and the definition $x = t$.

The new local contexts Γ_1 and Γ_2 that are produced through elaboration in Figures 5.4 and 5.5 are thrown away. This is intentional. New local contexts are produced during elaboration because the elaboration process can solve variables in the context through unification. However, the final resulting TT terms must again typecheck in only the global context.

If the resulting term was `Error` *err*, then elaboration of the program fails, reporting the error *err*. This error elaboration is defined in Figure 5.5. In this way, type providers can signal useful errors to users.

5.5 Expressiveness and Safety

F# type providers have essentially unlimited freedom to define new types as well as to generate code. These new types need not respect the ordinary rules of the F# type system — for example, they can have circular inheritance or refer to nonexisting 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 classic generated code.

From a fundamentalist point of view, this should not concern us. When we accepted that types could vary based on arbitrary I/O computations, we gave up not only properties such as decidability of typechecking, but even basic assumptions such as determinism of typechecking. 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 we have lost all of our guarantees, further departure should not concern us.

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. [Sym+12] 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 Idris type providers are different. In contrast to F# type providers, Idris type providers cannot generate new datatype definitions. In general, they are also unable to generate new top-level pattern-matching definitions (though this can be done to some extent as a side-effect of elaborating a `case` expression).

This reduction in expressiveness is matched by an increase in safety relative to F#. Idris type providers, as designed here, can only generate TT terms, which may or may not be actual types. These terms are subjected to the ordinary typechecking process, so we are just as safe with an Idris type provider as we 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 Idris type providers is exactly the expressiveness of indexed datatypes in Idris. In other words, to the extent that a universe

in the sense of Chapter 4 can represent the type in question, Idris type providers can also represent it.

A key aspect of F#'s type providers' expressiveness is their support for what Syme et al. [Sym+12] call "design-time" aspects. By this, they mean that type providers must support the programming process through quality tools. Indeed, such tools are 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 Idris type providers are simply generating ordinary terms, we should expect that the "design-time" support for them will be similar to that for other Idris programs.

5.6 Summary

This chapter has demonstrated the semantics of our proposed type provider mechanism for Idris. This extension can be defined by a straightforward extension of the elaboration mechanism. Idris type providers have two key advantages over F# type providers:

- Because they must always generate well-typed terms, they maintain the internal consistency of the type system.
- They do not need access to a special, internal API. Only ordinary Idris terms are used.

Because Idris type providers are ordinary programs in the `IO` monad, they can be developed and tested just like any other Idris program. Additionally, due to their straightforward semantics, there is no requirement that they be defined in one module and executed in another. However, the safety of Idris type providers has a downside: they have less expressive power than F# type providers. What remains to be examined is the expressiveness of the combination of dependent types and this simple feature. Chapters 8 and 9 explore example type providers, and Chapter 6 explores alternative definitions of type providers for a dependently-typed functional language.

Chapter 6

Design Considerations

Having read Chapter 5, one might wonder whether the presented design 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.

6.1 Top Level vs Expression Level

Idris type providers can only be used to generate top-level definitions. For example, one might wish to restrict the scope of a provided type using a `let` binding, as in Listing 6.1. 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.

There are two reasonable choices for the semantics of expression-level type providers: either we execute every provider with free vari-

```
let db = provide (loadSchema "some database")
in query db "..."
```

Listing 6.1: Hypothetical expression-level type provider

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

Listing 6.2: Hypothetical expression-level type provider with free variable

ables at runtime, when the values are known, or we make it a static error to refer to dynamic values. The former choice is not particularly satisfying, as the type provider mechanism would be equivalent to Haskell's `unsafePerformIO`. On the other hand, the latter choice is also not particularly satisfying. It should be possible to perform a binding-time analysis [JSS89] 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 uncontrolled 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.

6.2 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 Idris type providers.

However, this approach suffers from a 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. Second, it makes it possible for compile-time effects to "hide" in other code, with users unaware that the effects are occurring. The top-level type provider syntax that resembles a compiler directive signals that something potentially dangerous will occur.

6.3 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 lan-

guage in which type providers could be defined. The key insight is that *input* is much more important that *output*. If we can remove the ability of a type provider to send angry email to your boss or to delete 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.

We have explored some alternative formulations of dependent type providers and found them wanting. Let us now turn our attention to the actual implementation of type providers in Idris.

Chapter 7

Implementation

This chapter describes our implementation of Idris type providers and argues for the particular choices made in the implementation. In contrast to F# type providers, Idris type providers do not expose any internal compiler API. Thus, the implementation is largely orthogonal to the rest of the Idris compiler.

7.1 Elaboration

As discussed in Chapter 5, the elaboration of type providers to TT is based heavily on the already-existing elaboration mechanisms. As such, the elaborator in Listing 7.1 is quite straightforward. The helper function `isTType` simply returns `True` if and only if its argument is the TT term `Type`. The helper function `isProviderOf` returns `True` if and only if its second argument is `IO (Provider` τ`)`, where τ is the first argument.

The elaborator first checks that type providers are enabled. Assuming that this is the case, it then elaborates its argument `ty`, which corresponds to T in Figures 5.4 and 5.5. The variable `ty'` corresponds to τ. The function `elabVal` elaborates and typechecks an expression, returning both the TT expression and its type. In this case, we simply check that `typ` is in fact `Type`. Next, we elaborate the type provider using `elabVal` and check that it can be a provider for `ty'`.

Thus far, there has been a direct correspondence between the inference rules in Figures 5.4 and 5.5 and the elaborator. However, we now depart slightly. The top-level type declaration is created by re-elaborating the initial type instead of re-using the elaborated type. This

```
-- | Elaborate a type provider
elabProvider :: ElabInfo -> SyntaxInfo -> FC -> Name -> PTerm -> PTerm -> Idris ()
elabProvider info syn fc n ty tm
    = do i <- getIState
         -- Ensure that the experimental extension is enabled
         unless (TypeProviders `elem` idris_language_extensions i) $
           fail $ "Failed to define type provider \"" ++ show n ++
                  "\".\nYou must turn on the TypeProviders extension."

         ctxt <- getContext

         -- First elaborate the expected type (and check that it's a type)
         (ty', typ) <- elabVal toplevel False ty
         unless (isTType typ) $
           (fail $ "Expected a type, got " ++ show ty' ++ " : " ++ show typ)

         -- Elaborate the provider term to TT and check that the type matches
         (e, et) <- elabVal toplevel False tm
         unless (isProviderOf ty' et) $
           fail $ "Expected provider type IO (Provider (" ++
                  show ty' ++ "))" ++ ", got " ++ show et ++ " instead."

         -- Create the top-level type declaration
         elabType info syn "" fc [] n ty

         -- Execute the type provider and normalise the result
         rhs <- execute e
         let rhs' = normalise ctxt [] rhs

         -- Extract the provided term from the type provider
         tm <- getProvided rhs'

         -- Finally add a top-level definition of the provided term
         elabClauses info fc [] n [PClause fc n (PRef fc n) [] (delab i tm) []]
```

Listing 7.1: Elaborator for type providers

is simply to take advantage of the existing code in the Idris compiler for updating the global context in as straightforward a manner as possible.

Now that we have type-checked the type provider and created the global declaration, we execute the provider term. The result of execution is sent to the function `getProvided`, which simply checks whether the provider result is a success, a failure, or some other kind of term. If the result is a failure, then the error message is sent onward to the user using the ordinary error facilities of the `Idris` monad. This corresponds to the conclusion in Figure 5.5. If it is a success, then the resulting term is extracted. If it is neither a success nor a failure, then execution got "stuck", and a generic error is signalled.

Assuming that `getProvided` returned a term, the term is then de-elaborated to high-level Idris in order to again make use of the standard mechanisms for adding top-level Idris definitions. This ensures that the

definition is checked against the declared type once more, ensuring that a bug in the provider mechanism will not undermine the type safety of the existing system. The calls to `elabType` and `elabClauses` correspond roughly to the conclusion of Figure 5.4. They add the top-level type declaration and the definition of the provided term to the top-level definition context.

7.2 Execution

Idris type providers require the ability to execute arbitrary actions while elaborating and typechecking terms and definitions. However, Idris did not previously include an interpreter with this capability. Idris's original evaluator is written to make typechecking straightforward and efficient, not to be a practical interpreter for the full Idris language. Additionally, while most of the Idris compiler is written inside of a "kitchen sink" monad that includes both a large state and `IO`, the evaluator is outside this monad, making it easier to ensure its correctness.

It might be tempting to use the evaluator to find a normal form for a term, and then simply check whether that normal form is something that can be executed, possibly performing a side effect and then yielding a new term, which could be normalized and again executed, and so forth, until a non-I/O-action normal form was reached. However, this strategy is impractical. Reducing under lambdas, as the Idris evaluator must do, does not work well with precise ordering of IO actions. Even worse, normalizing an I/O term can cause unexecuted I/O actions to be duplicated. Thus, we need a separate evaluator to correctly run effectful programs, which we refer to as the *executor* to distinguish it from the evaluator used during typechecking.

We have developed such an executor. The full source code can be found in Appendix A. In general, it is a straightforward strict evaluator, with optional laziness implemented by generating thunks. The result of evaluation is a variant of the TT term language with a few additions:

- Higher-order abstract syntax [PE88] is used to simplify the scoping rules of binders
- New data constructors are added to represent thunks, C pointers, and file handles

Unlike the evaluator, the executor aims to be completely consistent with compiled code. It must respect optional laziness and strictly enforce the correct order of evaluation. Furthermore, it must actually perform I/O actions, so it is written within the Haskell `IO` monad.

To be useful, type providers need to interact with libraries that are not written in Idris. As C is the *lingua franca* for language interoperability, and Idris has a C FFI, the executor must be able to interact with C libraries. LibFFI[1] is used to implement Idris's FFI, where the interface to C libraries is specified as a datatype.

Additionally, a number of FFI calls are intercepted in the evaluator. This is because compiled Idris code is expected to be linked with a run-time system written in C. Many primitive operations and standard-library functions expect these run-time systems. However, for reasons of compatibility between 32-bit Haskell implementations on Mac OS X and 64-bit native libraries, it is not practical to simply dynamically load the run-time system.

Execution occurs in the `Exec` monad, which is a monad with I/O actions, a state and failure. The state contains certain global information from the `Idris` monad as well as information about thunks. Running execution in a separate monad allows us to isolate execution effects from the rest of the Idris system. Additionally, it makes the dependencies between the executor and the rest of the Idris compiler explicit.

[1] `http://sourceware.org/libffi/`

Part IV

Evaluation and Conclusion

Chapter 8

CSV Type Provider

Idris type providers aim to be a practical feature that can be used to solve the kinds of problems that programmers encounter daily. The first step towards assessing whether it succeeds in this aim is to actually use it. Therefore, the following two chapters describe two non-trivial type providers that interact with commonly used data formats.

Our first non-trivial demonstration of Idris type providers is a library for statically-checked access to comma-separated value (CSV) files. The CSV "format" is not completely standardized. Generally, it can be understood as a family of simple data formats, where each line in a text file is taken to be a row in a table, and some special delimiter character (usually comma, semicolon, or tab) is used to separate the columns. Each row is normally assumed to have the same number of columns. In some cases, the first row of the file contains column names. Conventions for escaping the delimiter character when it is a part of the data vary.

The CSV type provider works with a particular family of CSV dialects. In particular, arbitrary characters are allowed as delimiters, the first line of the file is assumed to contain column headers, and no provision is made for escaping delimiters as part of the data.

At compile-time, the CSV type provider requires an example of the CSV format that a program will read. The type provider then checks how many columns are present and reads their titles from the first line. Then, access to records from either the example file or one with the same format can be statically checked. It is possible to access the fields of a record either by column name or by column index.

```
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)
```

Listing 8.1: Named vectors

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

Listing 8.2: Bounded numbers (finite sets)

8.1 Named Vectors

CSV rows are represented by a variation of `Vect` called `NamedVect`. A `NamedVect` is just a `Vect` whose type contains an additional vector of `String`s that represent column names. It is possible to extract elements either by name or by index, and both are statically checked.

Listing 8.1 demonstrates the definition of named vectors. The names are simply implicit arguments, as they should be inferred from a type annotation.

8.1.1 Numeric lookup

In Section 4.1, we saw that the `Nat` index on the `Vect` datatype allows us to statically check that the length of the concatenation of two vectors is the sum of the length of the inputs. We can also use this information about the length of a vector to statically perform bounds-checking at lookup. This could be done by formulating a proposition that represents that one natural number is less than another and then requiring this as an argument to the lookup operation. However, it is perhaps more straightforward to simply define a type representing natural numbers with some upper bound. This is a standard example of dependent types, described accessibly by McBride [McB05].

The type `Fin n` has exactly `n` elements. By examining the indices on the return types of the constructors of `Fin`, we know that `Fin 0` is uninhabited: both indices invoke the `S` constructor. Any non-zero `Nat`

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

Listing 8.3: Vector lookup

will have the form `S n'` for some `n'`. We know that `f0` can construct an element of `Fin (S n')` — simply set the implicit argument `k` to `n'`. The constructor `fS` requires an argument that is in `Fin k`. It returns an element of `Fin (S k)`. Ignoring the indices, `Fin` has a quite similar structure to `Nat`. The implicit argument `k` is essentially representing the difference between the upper bound in the type and the `Nat`-like object constructed from `f0` and `fS`, and it is typically found automatically through unification.

The numeric lookup function `index` in Listing 8.3 uses the `Fin` datatype. The corresponding operation on traditional, unnamed `Vects` is essentially the same, with the only difference being in the type annotation. Because we know that there are exactly `n` inhabitants of `Fin n`, we have a one-to-one correspondence between it and the elements of our `n`-length named vector. Note that we do not have a case for the empty vector. This is because `Fin 0` is uninhabited, so we need not consider the argument whose type is `NamedVect a 0 ss`.

Assuming that we can convert rows from our CSV file into elements of `NamedVect`, we now have a means of extracting a particular column in a type-safe manner. However, this is not as convenient as we might wish — the column names are much more informative to those who may read the code. Luckily, our `NamedVect` also contains these.

8.1.2 Named lookup

Implementing a safe lookup by name is somewhat more complicated than by column index. We must be able to statically check that a name is, in fact, a member of the name vector, and we must have a means of extracting the corresponding value. We can satisfy both of these concerns by constructing a proof that the desired column is available in the name vector in such a way that the structure of the proof can be used to select the corresponding column.

Recall the type `Elem` from Listing 4.15. We can use a very similar type here. However, strings in Idris are primitives, which means that

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

Listing 8.4: Lookup by name

their structure is not available for inspection. Therefore, we cannot implement decidable equality, and we are forced to use Boolean equality, which is available as a primitive. In order to have decidability for our membership proof, we need to use an explicit representation of Boolean equality instead of relying on unification for the `Here` constructor. The type `so p` is inhabited if the expression `p` evaluates to `True`. The constructor `Here` then requires an element of `so (s == s')`, yielding that `Elem (s :: ss) s'`. Using the built-in Boolean equality primitive for strings, we can construct a decision procedure for `Elem` similar to that in Section 4.4.4.

It is straightforward to define a lookup function that uses the recursive structure of the proof of name membership to extract the corresponding element. Simply treat the proof as if it were a `Nat`, recursively calling the lookup function until the `Head` constructor is reached. The function `lookup'` in Listing 8.4 demonstrates this technique.

However, `lookup'` is not particularly convenient. Users need to construct the proof of membership each time. Even though there is a decision procedure that will construct the proof, user code will be littered with calls to this decision procedure and pattern-matches on its result. For use in a type provider, what we really want is a concise means of extracting columns by name and a safe means of conveniently ensuring that users do not mistype a column name. If they do misspell a column name, then typechecking should simply fail, rather than generating an error case that user code must handle.

Listing 8.5 demonstrates a more convenient wrapper around `lookup'` that automates the construction of the proof. An implicit argument declared with the `auto` keyword will be solved by applying `refl` — that is, it must have an equality type. The presence of the argument `x` simply causes `prf` to be solved by executing the decision procedure `decElem` and then unifying the result with `Yes prf`. If the decision procedure finds the proof then our straightforward `lookup'` is called. If it does not, then the type checker will fail with a unification 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' nv s p
```

Listing 8.5: Convenient lookup by name

```
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
```

Listing 8.6: CSV metadata

Now that we have a representation for the rows drawn from a CSV file, we can define a type provider.

8.2 CSV Types

To read our CSV files, we need to know what the delimiter character is, how many columns to expect, and the headers to apply to each column. Listing 8.6 demonstrates a straightforward representation of these facts. The datatype `CSVType` simply holds the relevant metadata, while `Row` provides a means of interpreting the `CSVType` as a `NamedVect`.

Our CSV type provider is therefore just a function to produce the `CSVType` that corresponds to a particular CSV format, given an example file. Listing 8.7 demonstrates an actual type provider for the supported dialect of CSV. It consists of just three simple functions. The function `cols` splits a string according to some delimiter character and removes leading and trailing whitespace from the results. The function `inferCSVType` takes the first line of some file that has been split with `cols` and constructs a `CSVType` instance. Finally, `csvType` uses these tools to read the first line of a file and construct a `CSVType`, failing if this

```
cols : Char -> String -> List String
cols delim row = map trim (split (==delim) row)

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

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
```

Listing 8.7: The CSV type provider

```
readCSVFile : (t : CSVType) -> String ->
              IO (Maybe (List (Row t)))
readCSVFile t file =
  do lines <- readLines file
     case lines of
       [] => return Nothing
       (h::body) =>
         if inferCSVType (delim t) h == t
           then return (Just (mapMaybe (readRow t)
                             (drop 1 lines)))
           else return Nothing
```

Listing 8.8: Reading a CSV file

is not possible.

When we actually use our type provider, we probably want some assurance that the files we open in fact correspond to the CSV type. Listing 8.8 demonstrates a means of opening a CSV file, but failing if the file does not match the desired schema. The function `readLines` in the body of `readCSVFile` simply returns a list of lines from the file as a string. The function `readRow t r` attempts to convert `r` to `Row t`, returning `Nothing` on failure.

```
Name ; Age ; Description ; Stuff
David ; 28 ; Working on thesis ; 23.2
Gracie ; 2 ; Young dog ; Playful
Cthulhu ; Innumerable eons ; Sleeping ; Ia ia!
```

Listing 8.9: The file “test.csv”

```
%provide (t : CSVType) with csvType ';' "test.csv"

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

partial
main : IO ()
main = do f <- readCSVFile t "test.csv"
          case f of
            Nothing => putStrLn "Couldn't open CSV file"
            Just rows =>
              do let ages = map getAge rows
                 let names = map (NamedVect.index f0) rows
                 putStrLn (show ages)
                 putStrLn (show names)
```

Listing 8.10: Using the CSV type provider

```
[28, 2, Innumerable eons]
[David, Gracie, Cthulhu]
```

Listing 8.11: Output from Listing 8.10

8.3 Example

Listing 8.9 demonstrates an example CSV file. Listing 8.10 uses the CSV type provider to read this file. First, `t` is bound to the inferred CSV type from the file in “test.csv”. It extracts the age column by name and the name column by numeric index, printing both to the screen.

This CSV type provider has been quite straightforward to define. However, it has a number of limitations. First, types of columns are not inferred — the programmer is stuck treating every column as a `String`.

In contrast, its F# counterpart[1] supports type annotations in the headers of CSV files and inference of types from column data. There is nothing preventing these features from being implemented in Idris, however. In fact, the next section describes a database interaction library with similar features. Second, the method used to implement the function `lookup` can be somewhat fragile. The function `getAge` is a top-level function only to convince Idris to solve implicit arguments, which it does not always do in the case of partially-applied functions. This is, however, a general limitation of Idris, and improvements in this area will improve the usefulness of type providers along with other, similar APIs.

[1] Available from `http://fsharp.github.io/FSharp.Data/library/CsvProvider.html`

Chapter 9

SQLite Type Provider

Our second extended example consists of a statically-checked database library, accessing the SQLite database. This is more substantial demonstration of the expressiveness of Idris type providers. The implementation is based on the technique in Oury and Swierstra's paper from ICFP 2008 [OS08].

Our SQLite type provider consists of a number of components:

- A library for representing database schemas and queries, with relational algebra as the underlying model
- A means of reading database schemas from SQLite files
- A library for interacting with SQLite
- A type provider built using these components

SQLite[1] is a library that provides a SQL API to local data. It is used internally in high-profile projects such as Firefox and Android. This project uses it due to its simplicity — the employed techniques are equally applicable to full-featured RDBMSs. Even though SQLite is to an extent dynamically typed, this type provider uses it as if it were statically typed.

9.1 Representing Data

The SQLite type provider relies on the following concepts, loosely based on the relational algebra:

[1] `http://www.sqlite.org`

Values are individual, scalar data points to be stored in the database, classified into a number of types;

Rows are ordered collections of values, addressable by index or by name;

Tables are are collections of rows;

Attributes are name/type pairs; and

Schemas are ordered collections of attributes.

Both rows and tables are classified by schemas: a row has a schema if its positionally corresponding values are classified by the types in the schema. A table is classified by a schema if all of its component rows are classified by the schema.

9.1.1 Types and Values

The SQLite type provider supports four basic datatypes: integers, text, real numbers, and Booleans. Additionally, the type provider supports both nullable and non-nullable versions of these types. This fact is represented through the simple universe (in the sense of Section 4.5) in Listing 9.1. It is straightforward to define the usual operations on `SQLType`, such as decidable equality and Boolean equality, though some care is necessary when defining Boolean equality on values drawn from the universe due to the interaction of type class resolution and dependent types. Ordinary Idris values represent the values drawn from the database.

While attributes could easily be represented as a pair of a string and a `SQLType`, better syntax and error messages can be obtained by defining a separate type. Schemas are defined such that Idris's syntax overloading can be used to represent them with a convenient list-like syntax. These types can be seen in Listing 9.2. The infix constructor `(:::)` binds more tightly than `(::)`, so `c ::: t :: s` is equivalent to `(c ::: t) :: s`.

Rather than defining the correspondence between rows and schemas as a separate predicate, it is convenient to index the datatype representing the row by the schema that should classify it. The function `interpSql` is used to relate the type in the schema to the value that can be stored in the row. We define a table by similarly indexing a list-like structure of rows. Again, these structures can be seen in Listing 9.2.

```
data SQLType = INTEGER
             | TEXT
             | NULLABLE SQLType
             | REAL
             | BOOLEAN

interpSql : SQLType -> Type
interpSql INTEGER = Int
interpSql TEXT = String
interpSql (NULLABLE t) = Maybe (interpSql t)
interpSql REAL = Float
interpSql BOOLEAN = Bool
```

Listing 9.1: A universe of database values

```
data Attribute : Type where
  (:::) : String -> SQLType -> Attribute

data Schema : Type where
  Nil : Schema
  (::) : (a : Attribute) -> (s : Schema) -> Schema

data Row : Schema -> Type where
  Nil : Row []
  (::) : (interpSql t) -> Row s -> Row (col ::: t :: s)

data Table : Schema -> Type where
  Nil : Table s
  (::) : Row s -> Table s -> Table s

Database : Type
Database = List (String, Schema)
```

Listing 9.2: Attributes

We now have a way to specify a schema and, by construction, ensure that a table and its corresponding rows and values conform to the schema. We define the type `Database` to simply be a list of pairs of table names and schemas. This is because queries must be made in the context of a number of tables.

9.2 Representing Queries

We represent queries as a deeply-embedded language that roughly corresponds to the relational algebra. In addition to references to tables, the query language has the following operations:

Union produces a table with all of the rows from both of its arguments;

Cartesian product concatenates each row of its first argument with each row of its second argument;

Projection extracts particular attributes from all rows in a table, forgetting the others;

Selection extracts the rows from a table that match some condition; and

Renaming changes the name of an attribute.

This section describes the query language, its type system, and its encoding in Idris.

There are certain obvious restrictions on these query operations to ensure that they are meaningful. For example, the union of two tables only makes sense if they both have the same schema and the schema of the result of a Cartesian product is the concatenation of its arguments' schemas. Some of the above properties are straightforward to encode by indexing the query datatype by a schema.

Let T range over the value types $\mathbb{Z}$, Text, $\mathbb{R}$, $\mathbb{B}$, and Nullable(T). An attribute consisting of a column name c and a value type T is written $c :: T$. Let a range over attributes. A sequence of attributes $a_1 \cdots a_n$ comprises a schema, written s.

Figure 9.1 demonstrates the syntax of queries. We have two typing judgments for queries and expressions, seen in Figure 9.2. Typing of queries is relative to some database from which the tables are drawn, and typing of expressions is relative to the schema of the row in which they are to be evaluated. The exact specification of expressions e would

t	The table t
$q_1 \cup q_2$	The union of q_1 and q_2
$q_1 \times q_2$	The Cartesian product of q_1 and q_2
$\pi_{a_1 \cdots a_n} q$	The projection of attributes $a_1, \ldots, a_n$ from query q
$\sigma_e q$	The selection of rows in which e is true from q
$\rho_{c_1 \to c_2} q$	Renaming column c_1 to c_2 in q

Figure 9.1: Queries

$d \vdash q : s$	Query q has schema s in database d
$s \vdash_E e : T$	Expression e has type T relative to schema s

Figure 9.2: Query judgments

take up too much space in this chapter — please refer to Appendix C for specifics.

The typing rules for queries contain a number of side conditions, such as the disjointness constraint on the Cartesian product rule and the requirement that projected attributes occur in the schema of the query from which they are projected. Fortunately, these side conditions are straightforward to encode as datatypes representing propositions.

Listing 9.3 demonstrates the key structure for representing queries. The type `Query` has a database as a parameter and a schema as an index, straightforwardly representing the typing rules in Figure 9.3. The type `Database` is merely a list of pairs of table names and their schemas. The side conditions in the typing rules are represented as implicit arguments. In the case of the Cartesian product, the actual structure of the proof is not needed, so the fact that it was found is sufficient. In the other cases, the compilation and interpretation of the query rules will make use of the proof structures of the side conditions. Thus, they are included as implicit arguments. Performing the same trick with an `auto` argument as was done with `lookup` in Listing 8.5 would be impractical, doubling the number of implicit arguments necessary. Instead, custom proof tactics are applied to construct the evidence. This technique also has the potential of providing better error messages in the future, once Idris-defined tactics can return custom error messages. This is an expected development.

This particular representation of queries is far from the only possibility. For example, the requirement that column names be unique could be

$$\frac{t \ : \ s \in d}{d \vdash t \ : \ s}$$

$$\frac{d \vdash q_1 \ : \ s \quad d \vdash q_2 \ : \ s}{d \vdash q_1 \cup q_2 \ : \ s}$$

$$\frac{d \vdash q_1 \ : \ s_1 \quad d \vdash q_2 \ : \ s_2 \quad s_1, s_2 \text{ disjoint}}{d \vdash q_1 \times q_2 \ : \ s_1 s_2}$$

$$\frac{d \vdash q \ : \ s \quad a_1 \in s \quad \cdots \quad a_n \in s}{d \vdash \pi_{a_1 \cdots a_n} q \ : \ a_1 \cdots a_n}$$

$$\frac{s \vdash_E e \ : \ \mathbb{B} \quad d \vdash q \ : \ s}{d \vdash \sigma_e q \ : \ s}$$

$$\frac{d \vdash q \ : \ a_1 \cdots a_n (c_1 ::: T) a_{n+2} \cdots a_m \quad c_2 \notin a_1 \cdots a_n (c_1 ::: T) a_{n+2} \cdots a_m}{d \vdash \rho_{c_1 \to c_2} q \ : \ a_1 \cdots a_n (c_2 ::: T) a_{n+2} \cdots a_m}$$

Figure 9.3: Typing rules for queries

solved as in SQL, where duplicate column names can be disambiguated by the name of their origin table. It would also be possible to construct a monadic interface in the style of LINQ [MBB06], or to construct a query language with a more SQL-like syntax. This particular representation was chosen because it is relatively simple and straightforward to understand.

9.3 The Type Provider

The entire purpose of a type provider is to ease interaction with external data sources. Thus, the database type provider must be able to read schemas from and execute queries on real databases. This is achieved through Idris's C FFI.

SQLite makes the schemas of its tables available by querying a metadata table. It is therefore straightforward to retrieve this information and convert it to a `Schema`. Our particular implementation of this conversion

```
data Query : Database -> Schema -> Type where
  T : (db : Database) -> (tbl : String) ->
      {default tactics {applyTactic findHasTable 50; solve;}
       ok : HasTable db tbl s} ->
      Query db s
  Union : Query db s -> Query db s -> Query db s
  Product : Query db s1 -> Query db s2 ->
            {default ()
             ok : isYes (decDisjoint s1 s2)} ->
            Query db (product s1 s2)
  Project : Query db s -> (s' : Schema) ->
            {default tactics {compute; applyTactic findSubSchema; solve;}
             ok : SubSchema s' s } ->
            Query db s'
  Select : Query db s -> Expr s BOOLEAN -> Query db s
  Rename : Query db s -> (from, to : String) ->
           {default tactics {compute; applyTactic findOccurs; solve;}
            fromOK : Occurs s from} ->
           {default tactics {compute; applyTactic findNotOccurs; solve;}
            toOK : Occurs s to -> _|_} ->
           Query db (Schemas.rename s from to fromOK toOK)
```

Listing 9.3: Queries

suffers from a hand-written parser that is not particularly readable. This is due to a current limitation in the Idris compiler that causes a code size explosion when using a combinator parser library in the style of Parsec [LM01].

The type provider, named `loadSchema`, simply queries SQLite for its schemas and returns their representation as a list of table names and `Schemas`. This can then be used as an index to the `Query` type.

It is straightforward to generate SQL queries from instances of `Query`. Each query operation has a direct parallel in SQL. Once the SQL has been generated, it is straightforward to use SQLite's C API to execute the query and extract the results.

Listing 9.5 demonstrates the use of the type provider. The SQLite file `test.sqlite` contains two tables: `people` and `transport`, the respective schemas of which can be seen in Listing 9.4.

The variable `query` represents a query that connects means of transportation with their owners. First, because the two tables each have a column called `id`, it must rename one of them. Next, it takes the Cartesian product of the two tables and selects those rows where `id` is equal to `owner`. The expression `Col "id" INTEGER` represents the contents of

```
CREATE TABLE "people" (
    "id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
    "name" VARCHAR NOT NULL,
    "age" INTEGER
);

CREATE TABLE "transport" (
    "id" INTEGER PRIMARY KEY NOT NULL UNIQUE,
    "owner" INTEGER NOT NULL,
    "wheels" INTEGER,
    "description" TEXT NOT NULL
);
```

Listing 9.4: Test database schema

id	name	age
1	'David'	28
2	'Hannes'	null

(a) The table `people`

id	owner	wheels	description
1	1	2	'bike'
2	2	2	'Recumbent'
3	2	1	'Unicycle'

(b) The table `transport`

Figure 9.4: Test data

the `Int`-typed column `"id"`. The annotation `INTEGER` is a limitation of the present implementation — it is checked, but not presently inferred. Finally, the desired columns are projected from the result.

9.4 Discussion

We now know that it is possible to construct a statically-checked relational database query library using Idris type providers. The present implementation is somewhat verbose. Additionally, the safety guarantees of the type theory are somewhat undermined by the use of C code to communicate with SQLite.

In their discussion of F# type providers, Syme et al. [Sym+12] use a notion of "conditional type soundness" — that is, type soundness up to the behavior of their type providers. This notion can be applied to Idris type providers that make use of unsafe features such as the C API. Additionally, we have no guarantee that the type system of the

```
%provide (testDB : Database) with loadSchema "test.sqlite"

-- Equivalent:  SELECT name, wheels, description
--              FROM people, transport
--              WHERE people.id = transport.owner

query : Query testDB [ "name"        ::: TEXT
                     , "wheels"      ::: NULLABLE INTEGER
                     , "description" ::: TEXT
                     ]
query = Project (Select (Product (T testDB "people")
                                 (Rename (T testDB "transport")
                                         "id" "transport_id"))
                        (Col "id" INTEGER == Col "owner" INTEGER))
                [ "name"        ::: TEXT
                , "wheels"      ::: NULLABLE INTEGER
                , "description" ::: TEXT
                ]

main : IO ()
main = do q <- doQuery "test.sqlite" query
          putStrLn (printTable q)
```

Listing 9.5: Demonstration of database type provider

```
name | wheels | description |
David | 2 | bike |
Hannes | 2 | Recumbent |
Hannes | 1 | Unicycle |
```

Listing 9.6: Output from test program

embedded query language actually corresponds to the type system of SQL in the correct manner.

From the perspective of conditional soundness, however, we can examine the correctness of client code under the assumption that the type provider successfully models its data source and is implemented correctly. In practice, we rely on large swaths of software that are not formally verified. As long as we recognize that our correctness hinges on the correct specification and implementation of the type provider, and take steps to ensure this correctness, we are no worse off than we are without the type providers.

Chapter 10

Evaluation

Idris type providers have the same goal as F# type providers: to conveniently enable statically-typed interaction with data sources that are defined outside of the language ecosystem. Recalling the Introduction, Idris type providers are a success to the extent that:

1. It is possible to derive a *safe* and *convenient* API from the schema of an external data source
2. It *supports very large schemas* in said external data sources
3. It is *easy to use*

We know that highly specific types are possible, due to Idris's full dependent types, easily fulfilling the safety criterion. However, because Idris type providers are unable to create new identifiers, we should expect that it may be more difficult to make this safe API convenient.

We have seen that it is possible to implement at least two substantial type providers using the Idris type provider mechanism. Because Idris type providers use the ordinary abstraction features of Idris, implementing a type provider ends up using many of the same techniques that are used when embedding a domain-specific language and its type system. More of these techniques could have been applied to the database library in particular to reduce the syntactic overhead of using it.

The process of developing the type providers revealed an important advantage of our technique: because type providers are ordinary Idris code, they can be developed using ordinary tools. The vast majority of the code supporting a type provider can be tested at the interactive toplevel. Additionally, because there is no need to define type providers

in a different module than the one in which they are used, it is possible to write a simple Idris file that contains a few test cases along with the type provider itself. While this is not a major benefit when programming "in the large", it is quite handy at the beginning of a project.

Idris type providers currently lack two key features of F# type providers: laziness and controlled erasure. It would not presently be possible to implement the Freebase type provider from Syme et al. [Sym+12], for instance, because the terms representing the schema would be too large. Likewise, there is not currently a precise control over type erasure. Because the Idris compiler performs aggressive type erasure during compilation, it may be possible to avoid very large compiled code as a result of large schemas. On the other hand, it is possible to achieve many of the same advantages as with F# type providers in providing a convenient, statically checked interface to externally-defined data.

It is not enough that a type provider mechanism makes it theoretically possible to define safe, convenient abstractions. It must also be possible for users to do this. In a short article on the Scala Web site,[1] Martin Odersky lays out a model of levels of Scala proficiency. The idea is that many users of the language can be productive without understanding all of the details, whereas library writers will often need to understand the language in depth. It may be the case that this model applies to dependently typed languages as well. If so, then the apparent complexity of implementing a convenient SQLite type provider should not worry us, as it only uses techniques that are common to other Idris libraries. Indeed, we hardly expect that every F# user will be skilled enough to define their own type providers. As long as advanced users can implement libraries that are usable by others, we can count Idris type providers as a success.

[1] `http://www.scala-lang.org/node/8610`

Chapter 11

Related Work

There are a large number of related techniques. F# type providers, being the inspiration for Idris type providers, are the most obvious related work. F# type providers are described in detail in Chapter 3, and compared to Idris type providers in Chapter 10.

Syme et al. [Sym+12] 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 Idris type providers.

11.1 Generic Programming with Universes

Using universes for dependently-typed generic programming has a long history. Benke, Dybjer, and Jansson [BDJ03] describe an early example in the framework of Martin-Löf type theory, and Altenkirch and McBride [AM03] provide a description that is perhaps easier to follow. More recently, Andjelkovic's M.Sc. thesis [And11] provides a large number of universes for generic programming, each supporting a different collection of features.

Idris 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 datatype-agnostic manner to achieve greater generality, Idris type providers seek to restrict the allowed programs to those that will work in a particular environment.

11.2 Ur/Web

Ur [Chl10] is a functional programming language that was designed to support a Web programming framework called Ur/Web. It features a highly expressive type system in which type-level records of types — that is, collections of pairs of names and types — can be transformed into actual record types. Additionally, Ur types can express conditions such as disjointness of record field names. Ur does not have dependent types. Type-level programming occurs in a separate class of terms. The aforementioned records of types use a separate record system from the one used by records of values.

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 metaprograms 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. [Sym+12] 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 from Chapters 8 and 9, 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 [Coq04] 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 [Agda] contains a reflection mechanism, described in van der Walt's M.Sc. thesis [Wal12], in which an Agda program can generate abstract syntax trees for Agda terms. The resulting terms are type-checked just as any other Agda terms. 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.

Chapter 12

Conclusion and Future Work

This thesis has demonstrated a mechanism, called *Idris type providers*, that can fulfill some of the goals of F#'s type providers, namely safe and convenient statically-typed access to external data sources. Work on Idris type providers is not finished. This chapter summarizes the work that has been done and suggests potentially fruitful avenues for future exploration.

Idris type providers have some advantages over F# type providers. Because they work entirely with ordinary terms, they are as safe as traditional code generation, while F# type providers can use their access to the internals of the compiler to generate illegal types and ill-typed terms. Their straightforward semantics 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 Idris 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. Additionally, because Idris 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. 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, subtyping is very expressive, and may provide opportunities that Idris lacks.

Idris 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.

12.1 Laziness and Erasure

Idris 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 erase provided types 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, McBride, and McKinna [BMM04] and Brady [Bra05]. 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.

12.2 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 [Wal12]. It would be interesting to extend Idris 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 prrof search tool (*e.g.* Z3 [MB08]) to find a term that inhabits the goal type.

12.3 FFI

While Haskell's C FFI 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.

12.4 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 convention 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.

Bibliography

[Agda] The Agda Team. *The Agda Wiki*. 2013. URL: `http://wiki.portal.chalmers.se/agda/`.

[AH05] David Aspinall and Martin Hofmann. "Dependent Types". In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin Pierce. MIT Press, 2005. Chap. 2.

[AM03] Thorsten Altenkirch and Conor McBride. "Generic programming within dependently typed programming". In: *Proceedings of IFIP TC2/WG2.1 Working Conference on Generic Programming*. 2003.

[And11] Stevan Andjelkovic. "A family of universes for generic programming". M.Sc. thesis. Chalmers University of Technology, 2011, p. 87.

[Aug98] Lennart Augustsson. "Cayenne — a language with dependent types". In: *SIGPLAN Notices* 34 (Sept. 1998), pp. 239–250.

[Bar91] Henk P. Barendregt. "Introduction to generalized type systems". In: *Journal of Functional Programming* 1.2 (Apr. 1991), pp. 125–154.

[BDJ03] Marcin Benke, Peter Dybjer, and Patrik Jansson. "Universes for generic programs and proofs in dependent type theory". In: *Nordic Journal of Computing* 10.4 (Dec. 2003), pp. 265–289.

[BMM04] Edwin Brady, Conor McBride, and James McKinna. "Inductive Families Need Not Store Their Indices". In: *Types for Proofs and Programs*. Ed. by Stefano Berardi, Mario Coppo, and Ferruccio Damiani. Vol. 3085. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 115–129.

[Bra05] Edwin Brady. “Practical Implementation of a Dependently Typed Functional Programming Language”. PhD thesis. University of Durham, 2005.

[Bra11] Edwin Brady. “Idris: systems programming meets full dependent types”. In: *Proceedings of the 5th ACM workshop on Programming languages meets program verification*. PLPV ’11. Austin, Texas, USA: ACM, 2011, pp. 43–54.

[Bra13] Edwin Brady. “Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation”. Draft of February 15, 2013. Under consideration for *Journal of Functional Programming*.

[Chl10] Adam Chlipala. “Ur: statically-typed metaprogramming with type-level record computation”. In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2010, pp. 122–133.

[Chr+14] David Raymond Christiansen, Henning Niss, Klaus Grue, Kristján S. Sigtryggsson, and Peter Sestoft. “An Actuarial Programming Language for Life Insurance”. Submitted to *International Congress of Actuaries, 2014*. 2014.

[Coq04] The Coq development team. *The Coq proof assistant reference manual*. Version 8.0. LogiCal Project. 2004.

[Coq92] Thierry Coquand. “Pattern Matching with Dependent Types”. In: *Electronic Proceedings of the Third Annual BRA Workshop on Logical Frameworks*. Ed. by Kent Petersson and Gordon Plotkin. Båstad, Sweden, 1992.

[EPJ01] Martin Erwig and Simon Peyton Jones. “Pattern Guards and Transformational Patterns”. In: *Electronic Notes in Theoretical Computer Science* 41.1 (2001).

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Gam+94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, Nov. 10, 1994.

[Har12] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. "Mix: A self-applicable partial evaluator for experiments in compiler generation". In: *LISP and Symbolic Computation* 2.1 (1989), pp. 9–50.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. "Strongly typed heterogeneous collections". In: *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: ACM, 2004, pp. 96–107.

[LM01] Daan Leijen and Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. UU-CS-2001-27. Department of Computer Science, Universiteit Utrecht, 2001.

[MB08] Leonardo Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C.R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 337–340.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. "LINQ: reconciling object, relations and XML in the .NET framework". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. Chicago, IL, USA: ACM, 2006, pp. 706–706.

[McB02] Conor McBride. "Faking it: Simulating dependent types in Haskell". In: *Journal of Functional Programming* 12 (4-5 July 2002), pp. 375–392.

[McB05] Conor McBride. "Epigram: Practical Programming with Dependent Types". In: *Advanced Functional Programming*. Ed. by Varmo Vene and Tarmo Uustalu. Vol. 3622. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 130–170.

[McB99] Conor McBride. "Dependently Typed Programs and their Proofs". Ph.D. Thesis. University of Edinburgh, 1999.

[Mil+97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. Rev Sub. The MIT Press, 1997.

[MM04] Conor McBride and James McKinna. "The view from the left". In: *Journal of Functional Programming* 14.1 (2004), pp. 69–111.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.

[Ode+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *An Overview of the Scala Programming Language*. Tech. rep. IC/2004/64. EPFL Lausanne, Switzerland, 2004.

[OMO10] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. "Type classes as objects and implicits". In: *SIGPLAN Notices* 45.10 (Oct. 2010), pp. 341–360.

[OS08] Nicolas Oury and Wouter Swierstra. "The power of Pi". In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*. Vol. 43. 9. New York, New York, USA: ACM Press, Sept. 2008, p. 39.

[PE88] F. Pfenning and C. Elliot. "Higher-order abstract syntax". In: *SIGPLAN Not.* 23 (1988), pp. 199–208.

[SII09] *Directive 2009/138/EC of the European Parliament and of the Council*.

[Sym+12] Don Syme et al. *Strongly-Typed Language Support for Internet-Scale Information Sources*. Tech. rep. MSR-TR-2012-101. Microsoft Research, Sept. 2012.

[Sym06] Don Syme. "Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution". In: *Proceedings of the 2006 workshop on ML*. Portland, Oregon, USA: ACM, 2006, pp. 43–54.

[Tur04] David Turner. "Total Functional Programming". In: *Journal of Universal Computer Science* 10 (7 2004), pp. 187–209.

[Wal12] Paul van der Walt. *Reflection in Agda*. M.Sc. Thesis. Available online at `http://igitur-archive.library.uu.nl/student-theses/2012-1030-200720/UUindex.html`. 2012.

[Yor+12] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dmitrios Vytiniotis, and José Pedro Magalhães. "Giving Haskell a promotion". In: *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*. TLDI '12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 53–66.

Glossary

code smell 31
A feature of source code that are not incorrect *per se*, but may indicate a deeper problem

decidable equality 33
An equality relation for which there exists a decision procedure

definitional equality 26
Equality arising from the equivalence rules in the type theory

dependent types 21
A type system in which types can quantify over terms as well as other types

elaboration 51
The process of explaining high-level programming language features in terms of a simpler core language

executor 63
An evaluator that is designed exclusively for executing I/O actions

non-linear patters 34
Patterns in which a the same name occurs more than once

principle of equivalence 6
The expected income and expenses for an insurance or pension product should be 0

propositional equality 27
A reification of the definition equality of the type system into a datatype

prospective statewise reserve 5
The expected net present value of the future payments due on an insurance or pension product

type provider 17
A compiler extension that is able to generate new types by executing arbitrary code

vector 22
In the dependent types literature, a list whose type precisely specifies its length

Appendix A

Executor

```
1  {-# LANGUAGE PatternGuards, ExistentialQuantification #-}
2  module Core.Execute (execute) where
3
4  import Idris.AbsSyntax
5  import Idris.AbsSyntaxTree
6  import IRTS.Lang( IntTy(..)
7                  , intTyToConst
8                  , FType(..))
9
10 import Core.TT
11 import Core.Evaluate
12 import Core.CaseTree
13
14 import Debug.Trace
15
16 import Util.DynamicLinker
17 import Util.System
18
19 import Control.Applicative hiding (Const)
20 import Control.Monad.Trans
21 import Control.Monad.Trans.State.Strict
22 import Control.Monad.Trans.Error
23 import Control.Monad
24 import Data.Maybe
25 import Data.Bits
26 import qualified Data.Map as M
27
28 import Foreign.LibFFI
29 import Foreign.C.String
30 import Foreign.Marshal.Alloc (free)
31 import Foreign.Ptr
32
33 import System.IO
34
35
36 data Lazy = Delayed ExecEnv Context Term | Forced ExecVal deriving Show
37
38 data ExecState = ExecState { exec_thunks :: M.Map Int Lazy -- ^ Thunks - the result of
       evaluating "lazy" or calling lazy funcs
39                            , exec_next_thunk :: Int -- ^ Ensure thunk key uniqueness
```

```
40                          , exec_implicits :: Ctxt [PArg] -- ^ Necessary info on laziness
                                from idris monad
41                          , exec_dynamic_libs :: [DynamicLib] -- ^ Dynamic libs from
                                idris monad
42                          }
43
44  data ExecVal = EP NameType Name ExecVal
45               | EV Int
46               | EBind Name (Binder ExecVal) (ExecVal -> Exec ExecVal)
47               | EApp ExecVal ExecVal
48               | EType UExp
49               | EErased
50               | EConstant Const
51               | forall a. EPtr (Ptr a)
52               | EThunk Int
53               | EHandle Handle
54
55  instance Show ExecVal where
56    show (EP _ n _)       = show n
57    show (EV i)           = "!!V" ++ show i ++ "!!"
58    show (EBind n b body) = "EBind " ++ show b ++ " <<fn>>"
59    show (EApp e1 e2)     = show e1 ++ " (" ++ show e2 ++ ")"
60    show (EType _)        = "Type"
61    show EErased          = "[__]"
62    show (EConstant c)    = show c
63    show (EPtr p)         = "<<ptr " ++ show p ++ ">>"
64    show (EThunk i)       = "<<thunk " ++ show i ++ ">>"
65    show (EHandle h)      = "<<handle " ++ show h ++ ">>"
66
67  toTT :: ExecVal -> Exec Term
68  toTT (EP nt n ty) = (P nt n) <$> (toTT ty)
69  toTT (EV i) = return $ V i
70  toTT (EBind n b body) = do body' <- body $ EP Bound n EErased
71                             b' <- fixBinder b
72                             Bind n b' <$> toTT body'
73      where fixBinder (Lam t)       = Lam   <$> toTT t
74            fixBinder (Pi t)        = Pi    <$> toTT t
75            fixBinder (Let t1 t2)   = Let   <$> toTT t1 <*> toTT t2
76            fixBinder (NLet t1 t2)  = NLet  <$> toTT t1 <*> toTT t2
77            fixBinder (Hole t)      = Hole  <$> toTT t
78            fixBinder (GHole t)     = GHole <$> toTT t
79            fixBinder (Guess t1 t2) = Guess <$> toTT t1 <*> toTT t2
80            fixBinder (PVar t)      = PVar  <$> toTT t
81            fixBinder (PVTy t)      = PVTy  <$> toTT t
82  toTT (EApp e1 e2) = do e1' <- toTT e1
83                         e2' <- toTT e2
84                         return $ App e1' e2'
85  toTT (EType u) = return $ TType u
86  toTT EErased = return Erased
87  toTT (EConstant c) = return (Constant c)
88  toTT (EThunk i) = return (P (DCon 0 0) (MN i "THUNK") Erased) --(force i) >>= toTT
89  toTT (EHandle _) = return Erased
90
91  unApplyV :: ExecVal -> (ExecVal, [ExecVal])
92  unApplyV tm = ua [] tm
93      where ua args (EApp f a) = ua (a:args) f
94            ua args t = (t, args)
95
96  mkEApp :: ExecVal -> [ExecVal] -> ExecVal
```

```
97  mkEApp f [] = f
98  mkEApp f (a:args) = mkEApp (EApp f a) args
99
100 initState :: Idris ExecState
101 initState = do ist <- getIState
102                return $ ExecState M.empty 0 (idris_implicits ist) (idris_dynamic_libs ist)
103
104 type Exec = ErrorT String (StateT ExecState IO)
105
106 runExec :: Exec a -> ExecState -> IO (Either String a)
107 runExec ex st = fst <$> runStateT (runErrorT ex) st
108
109 getExecState :: Exec ExecState
110 getExecState = lift get
111
112 putExecState :: ExecState -> Exec ()
113 putExecState = lift . put
114
115 execFail :: String -> Exec a
116 execFail = throwError
117
118 execIO :: IO a -> Exec a
119 execIO = lift . lift
120
121 delay :: ExecEnv -> Context -> Term -> Exec ExecVal
122 delay env ctxt tm =
123     do st <- getExecState
124        let i = exec_next_thunk st
125        putExecState $ st { exec_thunks = M.insert i (Delayed env ctxt tm) (exec_thunks st)
126                          , exec_next_thunk = exec_next_thunk st + 1
127                          }
128        return $ EThunk i
129
130 force :: Int -> Exec ExecVal
131 force i = do st <- getExecState
132              case M.lookup i (exec_thunks st) of
133                Just (Delayed env ctxt tm) -> do tm' <- doExec env ctxt tm
134                                                 case tm' of
135                                                   EThunk i ->
136                                                       do res <- force i
137                                                          update res i
138                                                          return res
139                                                   _ -> do update tm' i
140                                                           return tm'
141                Just (Forced tm) -> return tm
142                Nothing -> execFail "Tried to exec non-existing thunk. This is a bug!"
143     where update :: ExecVal -> Int -> Exec ()
144           update tm i = do est <- getExecState
145                            putExecState $ est { exec_thunks = M.insert i (Forced tm)
                                                      (exec_thunks est) }
146
147 tryForce :: ExecVal -> Exec ExecVal
148 tryForce (EThunk i) = force i
149 tryForce tm = return tm
150
151 debugThunks :: Exec ()
152 debugThunks = do st <- getExecState
153                  execIO $ putStrLn (take 4000 (show (exec_thunks st)))
154
```

```
155 execute :: Term -> Idris Term
156 execute tm = do est <- initState
157                 ctxt <- getContext
158                 res <- lift $ flip runExec est $ do res <- doExec [] ctxt tm
159                                                     toTT res
160                 case res of
161                   Left err -> fail err
162                   Right tm' -> return tm'
163
164 ioWrap :: ExecVal -> ExecVal
165 ioWrap tm = mkEApp (EP (DCon 0 2) (UN "prim__IO") EErased) [EErased, tm]
166
167 ioUnit :: ExecVal
168 ioUnit = ioWrap (EP Ref unitCon EErased)
169
170 type ExecEnv = [(Name, ExecVal)]
171
172 doExec :: ExecEnv -> Context -> Term -> Exec ExecVal
173 doExec env ctxt p@(P Ref n ty) | Just v <- lookup n env = return v
174 doExec env ctxt p@(P Ref n ty) =
175     do let val = lookupDef n ctxt
176        case val of
177          [Function _ tm] -> doExec env ctxt tm
178          [TyDecl _ _] -> return (EP Ref n EErased) -- abstract def
179          [Operator tp arity op] -> return (EP Ref n EErased) -- will be special-cased later
180          [CaseOp _ _ _ _ _ [] (STerm tm) _ _] -> -- nullary fun
181              doExec env ctxt tm
182          [CaseOp _ _ _ _ _ ns sc _ _] -> return (EP Ref n EErased)
183          [] -> execFail $ "Could not find " ++ show n ++ " in definitions."
184          thing -> trace (take 200 $ "got to " ++ show thing ++ " lookup up " ++ show n) $
                undefined
185 doExec env ctxt p@(P Bound n ty) =
186   case lookup n env of
187     Nothing -> execFail "not found"
188     Just tm -> return tm
189 doExec env ctxt (P (DCon a b) n _) = return (EP (DCon a b) n EErased)
190 doExec env ctxt (P (TCon a b) n _) = return (EP (TCon a b) n EErased)
191 doExec env ctxt v@(V i) | i < length env = return (snd (env !! i))
192                         | otherwise      = execFail "env too small"
193 doExec env ctxt (Bind n (Let t v) body) = do v' <- doExec env ctxt v
194                                              doExec ((n, v'):env) ctxt body
195 doExec env ctxt (Bind n (NLet t v) body) = trace "NLet" $ undefined
196 doExec env ctxt tm@(Bind n b body) = return $
197                                      EBind n (fmap (\_->EErased) b)
198                                            (\arg -> doExec ((n, arg):env) ctxt body)
199 doExec env ctxt a@(App _ _) = execApp env ctxt (unApply a)
200 doExec env ctxt (Constant c) = return (EConstant c)
201 doExec env ctxt (Proj tm i) = let (x, xs) = unApply tm in
202                               doExec env ctxt ((x:xs) !! i)
203 doExec env ctxt Erased = return EErased
204 doExec env ctxt Impossible = fail "Tried to execute an impossible case"
205 doExec env ctxt (TType u) = return (EType u)
206
207 execApp :: ExecEnv -> Context -> (Term, [Term]) -> Exec ExecVal
208 execApp env ctxt (f, args) = do newF <- doExec env ctxt f
209                                 laziness <- (++ repeat False) <$> (getLaziness newF)
210                                 newArgs <- mapM argExec (zip args laziness)
211                                 execApp' env ctxt newF newArgs
212     where getLaziness (EP _ (UN "lazy") _) = return [False, True]
```

```
213           getLaziness (EP _ n _) = do est <- getExecState
214                                       let argInfo = exec_implicits est
215                                       case lookupCtxtName n argInfo of
216                                         [] -> return (repeat False)
217                                         [ps] -> return $ map lazyarg (snd ps)
218                                         many -> execFail $ "Ambiguous " ++ show n ++ ",
                                                 found " ++ (take 200 $ show many)
219           getLaziness _ = return (repeat False) -- ok due to zip above
220           argExec :: (Term, Bool) -> Exec ExecVal
221           argExec (tm, False) = doExec env ctxt tm
222           argExec (tm, True) = delay env ctxt tm
223
224
225 execApp' :: ExecEnv -> Context -> ExecVal -> [ExecVal] -> Exec ExecVal
226 execApp' env ctxt v [] = return v -- no args is just a constant! can result from function
      calls
227 execApp' env ctxt (EP _ (UN "unsafePerformIO") _) (ty:action:rest) | (prim__IO, [_, v]) <-
      unApplyV action =
228     execApp' env ctxt v rest
229
230 execApp' env ctxt (EP _ (UN "io_bind") _) args@(_:_:v:k:rest) | (prim__IO, [_, v']) <-
      unApplyV v =
231     do v'' <- tryForce v'
232        res <- execApp' env ctxt k [v''] >>= tryForce
233        execApp' env ctxt res rest
234 execApp' env ctxt con@(EP _ (UN "io_return") _) args@(tp:v:rest) =
235     do v' <- tryForce v
236        execApp' env ctxt (mkEApp con [tp, v']) rest
237
238 -- Special cases arising from not having access to the C RTS in the interpreter
239 execApp' env ctxt (EP _ (UN "mkForeign") _) (_:fn:EConstant (Str arg):rest)
240     | Just (FFun "putStr" _ _) <- foreignFromTT fn = do execIO (putStr arg)
241                                                         execApp' env ctxt ioUnit rest
242 execApp' env ctxt (EP _ (UN "mkForeign") _) (_:fn:EConstant (Str f):EConstant (Str
      mode):rest)
243     | Just (FFun "fileOpen" _ _) <- foreignFromTT fn = do m <- case mode of
244                                                                  "r" -> return ReadMode
245                                                                  "w" -> return WriteMode
246                                                                  "a" -> return AppendMode
247                                                                  "rw" -> return
                                                                       ReadWriteMode
248                                                                  "wr" -> return
                                                                       ReadWriteMode
249                                                                  "r+" -> return
                                                                       ReadWriteMode
250                                                                  _ -> execFail ("Invalid
                                                                       mode for " ++ f ++
                                                                       ": " ++ mode)
251                                                           h <- execIO $ openFile f m
252                                                           execApp' env ctxt (ioWrap
                                                                (EHandle h)) rest
253
254 execApp' env ctxt (EP _ (UN "mkForeign") _) (_:fn:(EHandle h):rest)
255     | Just (FFun "fileEOF" _ _) <- foreignFromTT fn = do eofp <- execIO $ hIsEOF h
256                                                         let res = ioWrap (EConstant (I $
                                                                 if eofp then 1 else 0))
257                                                         execApp' env ctxt res rest
258
259 execApp' env ctxt (EP _ (UN "mkForeign") _) (_:fn:(EHandle h):rest)
```

```
260     | Just (FFun "fileClose" _ _) <- foreignFromTT fn = do execIO $ hClose h
261                                                          execApp' env ctxt ioUnit rest
262
263 execApp' env ctxt (EP _ (UN "mkForeign") _) (_:fn:(EPtr p):rest)
264     | Just (FFun "isNull" _ _) <- foreignFromTT fn = let res = ioWrap . EConstant . I $
265                                                                if p == nullPtr then 1 else
                                                                     0
266                                                    in execApp' env ctxt res rest
267
268 execApp' env ctxt f@(EP _ (UN "mkForeign") _) args@(ty:fn:xs) | Just (FFun f argTs retT)
      <- foreignFromTT fn
269                                                                , length xs >= length argTs =
270     do res <- stepForeign (ty:fn:take (length argTs) xs)
271        case res of
272          Nothing -> fail $ "Could not call foreign function \"" ++ f ++
273                            "\" with args " ++ show (take (length argTs) xs)
274          Just r -> return (mkEApp r (drop (length argTs) xs))
275                                                              | otherwise = return (mkEApp
                                                                    f args)
276
277 execApp' env ctxt f@(EP _ n _) args =
278     do let val = lookupDef n ctxt
279        case val of
280          [Function _ tm] -> fail "should already have been eval'd"
281          [TyDecl nt ty] -> return $ mkEApp f args
282          [Operator tp arity op] ->
283              if length args >= arity
284                then do args' <- mapM tryForce $ take arity args
285                        case getOp n args' of
286                          Just res -> do r <- res
287                                         execApp' env ctxt r (drop arity args)
288                          Nothing -> return (mkEApp f args)
289                else return (mkEApp f args)
290          [CaseOp _ _ _ _ _ [] (STerm tm) _ _] -> -- nullary fun
291              do rhs <- doExec env ctxt tm
292                 execApp' env ctxt rhs args
293          [CaseOp _ _ _ _ _  ns sc _ _] ->
294              do res <- execCase env ctxt ns sc args
295                 return $ fromMaybe (mkEApp f args) res
296          thing -> return $ mkEApp f args
297     where getOp :: Name -> [ExecVal] -> Maybe (Exec ExecVal)
298           getOp (UN "prim__addInt") [EConstant (I i1), EConstant (I i2)] =
299               primRes I (i1 + i2)
300           getOp (UN "prim__andInt") [EConstant (I i1), EConstant (I i2)] =
301               primRes I (i1 .&. i2)
302           getOp (UN "prim__charToInt") [EConstant (Ch c)] =
303               primRes I (fromEnum c)
304           getOp (UN "prim__complInt") [EConstant (I i)] =
305               primRes I (complement i)
306           getOp (UN "prim__concat") [EConstant (Str s1), EConstant (Str s2)] =
307               primRes Str (s1 ++ s2)
308           getOp (UN "prim__divInt") [EConstant (I i1), EConstant (I i2)] =
309               primRes I (i1 `div` i2)
310           getOp (UN "prim__eqChar") [EConstant (Ch c1), EConstant (Ch c2)] =
311               primResBool (c1 == c2)
312           getOp (UN "prim__eqInt") [EConstant (I i1), EConstant (I i2)] =
313               primResBool (i1 == i2)
314           getOp (UN "prim__eqString") [EConstant (Str s1), EConstant (Str s2)] =
315               primResBool (s1 == s2)
```

```
316             getOp (UN "prim__gtChar") [EConstant (Ch i1), EConstant (Ch i2)] =
317                 primResBool (i1 > i2)
318             getOp (UN "prim__gteChar") [EConstant (Ch i1), EConstant (Ch i2)] =
319                 primResBool (i1 >= i2)
320             getOp (UN "prim__gtInt") [EConstant (I i1), EConstant (I i2)] =
321                 primResBool (i1 > i2)
322             getOp (UN "prim__gteInt") [EConstant (I i1), EConstant (I i2)] =
323                 primResBool (i1 >= i2)
324             getOp (UN "prim__intToFloat") [EConstant (I i)] =
325                 primRes Fl (fromRational (toRational i))
326             getOp (UN "prim__intToStr") [EConstant (I i)] =
327                 primRes Str (show i)
328             getOp (UN "prim_lenString") [EConstant (Str s)] =
329                 primRes I (length s)
330             getOp (UN "prim__ltChar") [EConstant (Ch i1), EConstant (Ch i2)] =
331                 primResBool (i1 < i2)
332             getOp (UN "prim__lteChar") [EConstant (Ch i1), EConstant (Ch i2)] =
333                 primResBool (i1 <= i2)
334             getOp (UN "prim__lteInt") [EConstant (I i1), EConstant (I i2)] =
335                 primResBool (i1 <= i2)
336             getOp (UN "prim__ltInt") [EConstant (I i1), EConstant (I i2)] =
337                 primResBool (i1 < i2)
338             getOp (UN "prim__modInt") [EConstant (I i1), EConstant (I i2)] =
339                 primRes I (i1 `mod` i2)
340             getOp (UN "prim__mulBigInt") [EConstant (BI i1), EConstant (BI i2)] =
341                 primRes BI (i1 * i2)
342             getOp (UN "prim__mulFloat") [EConstant (Fl i1), EConstant (Fl i2)] =
343                 primRes Fl (i1 * i2)
344             getOp (UN "prim__mulInt") [EConstant (I i1), EConstant (I i2)] =
345                 primRes I (i1 * i2)
346             getOp (UN "prim__orInt") [EConstant (I i1), EConstant (I i2)] =
347                 primRes I (i1 .|. i2)
348             getOp (UN "prim__readString") [EP _ (UN "prim__stdin") _] =
349                 Just $ do line <- execIO getLine
350                           return (EConstant (Str line))
351             getOp (UN "prim__readString") [EHandle h] =
352                 Just $ do contents <- execIO $ hGetLine h
353                           return (EConstant (Str contents))
354             getOp (UN "prim__shLInt") [EConstant (I i1), EConstant (I i2)] =
355                 primRes I (shiftL i1 i2)
356             getOp (UN "prim__shRInt") [EConstant (I i1), EConstant (I i2)] =
357                 primRes I (shiftR i1 i2)
358             getOp (UN "prim__strCons") [EConstant (Ch c), EConstant (Str s)] =
359                 primRes Str (c:s)
360             getOp (UN "prim__strHead") [EConstant (Str (c:s))] =
361                 primRes Ch c
362             getOp (UN "prim__strRev") [EConstant (Str s)] =
363                 primRes Str (reverse s)
364             getOp (UN "prim__strTail") [EConstant (Str (c:s))] =
365                 primRes Str s
366             getOp (UN "prim__subFloat") [EConstant (Fl i1), EConstant (Fl i2)] =
367                 primRes Fl (i1 - i2)
368             getOp (UN "prim__subInt") [EConstant (I i1), EConstant (I i2)] =
369                 primRes I (i1 - i2)
370             getOp (UN "prim__xorInt") [EConstant (I i1), EConstant (I i2)] =
371                 primRes I (i1 `xor` i2)
372             getOp n args = trace ("No prim " ++ show n ++ " for " ++ take 1000 (show args))
                  Nothing
373
```

```
374           primRes :: (a -> Const) -> a -> Maybe (Exec ExecVal)
375           primRes constr = Just . return . EConstant . constr
376
377           primResBool :: Bool -> Maybe (Exec ExecVal)
378           primResBool b = primRes I (if b then 1 else 0)
379
380 execApp' env ctxt bnd@(EBind n b body) (arg:args) = do ret <- body arg
381                                                         let (f', as) = unApplyV ret
382                                                         execApp' env ctxt f' (as ++ args)
383 execApp' env ctxt (EThunk i) args = do f <- force i
384                                        execApp' env ctxt f args
385 execApp' env ctxt app@(EApp _ _) args2 | (f, args1) <- unApplyV app = execApp' env ctxt f
       (args1 ++ args2)
386 execApp' env ctxt f args = return (mkEApp f args)
387
388
389
390
391
392 -- | Overall wrapper for case tree execution. If there are enough arguments, it takes them,
393 -- evaluates them, then begins the checks for matching cases.
394 execCase :: ExecEnv -> Context -> [Name] -> SC -> [ExecVal] -> Exec (Maybe ExecVal)
395 execCase env ctxt ns sc args =
396     let arity = length ns in
397     if arity <= length args
398     then do let amap = zip ns args
399 --            trace ("Case " ++ show sc ++ "\n   in " ++ show amap ++"\n   in env " ++
       show env ++ "\n\n" ) $ return ()
400             caseRes <- execCase' env ctxt amap sc
401             case caseRes of
402               Just res -> Just <$> execApp' (map (\(n, tm) -> (n, tm)) amap ++ env) ctxt
                    res (drop arity args)
403               Nothing -> return Nothing
404     else return Nothing
405
406 -- | Take bindings and a case tree and examines them, executing the matching case if
       possible.
407 execCase' :: ExecEnv -> Context -> [(Name, ExecVal)] -> SC -> Exec (Maybe ExecVal)
408 execCase' env ctxt amap (UnmatchedCase _) = return Nothing
409 execCase' env ctxt amap (STerm tm) =
410     Just <$> doExec (map (\(n, v) -> (n, v)) amap ++ env) ctxt tm
411 execCase' env ctxt amap (Case n alts) | Just tm <- lookup n amap =
412     do tm' <- tryForce tm
413        case chooseAlt tm' alts of
414          Just (newCase, newBindings) ->
415              let amap' = newBindings ++ (filter (\(x,_) -> not (elem x (map fst
                    newBindings))) amap) in
416              execCase' env ctxt amap' newCase
417          Nothing -> return Nothing
418
419 chooseAlt :: ExecVal -> [CaseAlt] -> Maybe (SC, [(Name, ExecVal)])
420 chooseAlt _ (DefaultCase sc : alts) = Just (sc, [])
421 chooseAlt (EConstant c) (ConstCase c' sc : alts) | c == c' = Just (sc, [])
422 chooseAlt tm (ConCase n i ns sc : alts) | ((EP _ cn _), args) <- unApplyV tm
423                                         , cn == n = Just (sc, zip ns args)
424                                         | otherwise = chooseAlt tm alts
425 chooseAlt tm (_:alts) = chooseAlt tm alts
426 chooseAlt _ [] = Nothing
427
```

```
428
429
430
431 idrisType :: FType -> ExecVal
432 idrisType FUnit = EP Ref unitTy EErased
433 idrisType ft = EConstant (idr ft)
434     where idr (FInt ty) = intTyToConst ty
435           idr FDouble = FlType
436           idr FChar = ChType
437           idr FString = StrType
438           idr FPtr = PtrType
439
440 data Foreign = FFun String [FType] FType deriving Show
441
442
443 call :: Foreign -> [ExecVal] -> Exec (Maybe ExecVal)
444 call (FFun name argTypes retType) args =
445     do fn <- findForeign name
446        case fn of
447          Nothing -> return Nothing
448          Just f -> do res <- call' f args retType
449                       return . Just . ioWrap $ res
450     where call' :: ForeignFun -> [ExecVal] -> FType -> Exec ExecVal
451           call' (Fun _ h) args (FInt ITNative) = do res <- execIO $ callFFI h retCInt
                  (prepArgs args)
452                                                    return (EConstant (I (fromIntegral
                                                         res)))
453           call' (Fun _ h) args (FInt IT8) = do res <- execIO $ callFFI h retCChar
                  (prepArgs args)
454                                               return (EConstant (B8 (fromIntegral res)))
455           call' (Fun _ h) args (FInt IT16) = do res <- execIO $ callFFI h retCWchar
                  (prepArgs args)
456                                                return (EConstant (B16 (fromIntegral res)))
457           call' (Fun _ h) args (FInt IT32) = do res <- execIO $ callFFI h retCInt
                  (prepArgs args)
458                                                return (EConstant (B32 (fromIntegral res)))
459           call' (Fun _ h) args (FInt IT64) = do res <- execIO $ callFFI h retCLong
                  (prepArgs args)
460                                                return (EConstant (B64 (fromIntegral res)))
461           call' (Fun _ h) args FDouble = do res <- execIO $ callFFI h retCDouble (prepArgs
                  args)
462                                             return (EConstant (Fl (realToFrac res)))
463           call' (Fun _ h) args FChar = do res <- execIO $ callFFI h retCChar (prepArgs
                  args)
464                                           return (EConstant (Ch (castCCharToChar res)))
465           call' (Fun _ h) args FString = do res <- execIO $ callFFI h retCString (prepArgs
                  args)
466                                             hStr <- execIO $ peekCString res
467 --                                            lift $ free res
468                                             return (EConstant (Str hStr))
469
470           call' (Fun _ h) args FPtr = do res <- execIO $ callFFI h (retPtr retVoid)
                  (prepArgs args)
471                                         return (EPtr res)
472           call' (Fun _ h) args FUnit = do res <- execIO $ callFFI h retVoid (prepArgs args)
473                                           return (EP Ref unitCon EErased)
474 --          call' (Fun _ h) args other = fail ("Unsupported foreign return type " ++ show
      other)
475
```

```
476
477           prepArgs = map prepArg
478           prepArg (EConstant (I i)) = argCInt (fromIntegral i)
479           prepArg (EConstant (B8 i)) = argCChar (fromIntegral i)
480           prepArg (EConstant (B16 i)) = argCWchar (fromIntegral i)
481           prepArg (EConstant (B32 i)) = argCInt (fromIntegral i)
482           prepArg (EConstant (B64 i)) = argCLong (fromIntegral i)
483           prepArg (EConstant (Fl f)) = argCDouble (realToFrac f)
484           prepArg (EConstant (Ch c)) = argCChar (castCharToCChar c) -- FIXME -
                  castCharToCChar only safe for first 256 chars
485           prepArg (EConstant (Str s)) = argString s
486           prepArg (EPtr p) = argPtr p
487           prepArg other = trace ("Could not use " ++ take 100 (show other) ++ " as FFI
                  arg.") undefined
488
489
490
491 foreignFromTT :: ExecVal -> Maybe Foreign
492 foreignFromTT t = case (unApplyV t) of
493                     (_, [(EConstant (Str name)), args, ret]) ->
494                         do argTy <- unEList args
495                            argFTy <- sequence $ map getFTy argTy
496                            retFTy <- getFTy ret
497                            return $ FFun name argFTy retFTy
498                     _ -> trace "failed to construct ffun" Nothing
499
500 getFTy :: ExecVal -> Maybe FType
501 getFTy (EApp (EP _ (UN "FIntT") _) (EP _ (UN intTy) _)) =
502     case intTy of
503       "ITNative" -> Just $ FInt ITNative
504       "IT8" -> Just $ FInt IT8
505       "IT16" -> Just $ FInt IT16
506       "IT32" -> Just $ FInt IT32
507       "IT64" -> Just $ FInt IT64
508       _ -> Nothing
509 getFTy (EP _ (UN t) _) =
510     case t of
511       "FFloat"  -> Just FDouble
512       "FChar"   -> Just FChar
513       "FString" -> Just FString
514       "FPtr"    -> Just FPtr
515       "FUnit"   -> Just FUnit
516       _         -> Nothing
517 getFTy _ = Nothing
518
519 unList :: Term -> Maybe [Term]
520 unList tm = case unApply tm of
521               (nil, [_]) -> Just []
522               (cons, ([_, x, xs])) ->
523                   do rest <- unList xs
524                      return $ x:rest
525               (f, args) -> Nothing
526
527 unEList :: ExecVal -> Maybe [ExecVal]
528 unEList tm = case unApplyV tm of
529                (nil, [_]) -> Just []
530                (cons, ([_, x, xs])) ->
531                    do rest <- unEList xs
532                       return $ x:rest
```

```
533                (f, args) -> Nothing
534
535
536  toConst :: Term -> Maybe Const
537  toConst (Constant c) = Just c
538  toConst _ = Nothing
539
540  stepForeign :: [ExecVal] -> Exec (Maybe ExecVal)
541  stepForeign (ty:fn:args) = do let ffun = foreignFromTT fn
542                                f' <- case (call <$> ffun) of
543                                        Just f -> f args
544                                        Nothing -> return Nothing
545                                return f'
546  stepForeign _ = fail "Tried to call foreign function that wasn't mkForeign"
547
548  mapMaybeM :: Monad m => (a -> m (Maybe b)) -> [a] -> m [b]
549  mapMaybeM f [] = return []
550  mapMaybeM f (x:xs) = do rest <- mapMaybeM f xs
551                          x' <- f x
552                          case x' of
553                            Just x'' -> return (x'':rest)
554                            Nothing -> return rest
555
556  findForeign :: String -> Exec (Maybe ForeignFun)
557  findForeign fn = do est <- getExecState
558                      let libs = exec_dynamic_libs est
559                      fns <- mapMaybeM getFn libs
560                      case fns of
561                        [f] -> return (Just f)
562                        [] -> do execIO . putStrLn $ "Symbol \"" ++ fn ++ "\" not found"
563                                 return Nothing
564                        fs -> do execIO . putStrLn $ "Symbol \"" ++ fn ++ "\" is ambiguous.
                                      Found " ++
565                                                                  show (length fs) ++ " occurrences."
566                                 return Nothing
567      where getFn lib = execIO $ catchIO (tryLoadFn fn lib) (\_ -> return Nothing)
```

Appendix B

CSV Type Provider

Data.NamedVect

```
1  module Data.NamedVect
2
3  import Decidable.Equality
4
5  soNot : so p -> so (not p) -> _|_
6  soNot oh oh impossible
7
8
9  -- | NamedVect a n ss is an n-length vector of as, named by the elements in ss
10 data NamedVect : Type -> (n : Nat) -> (Vect n String) -> Type where
11   Nil : NamedVect a Z []
12   (::) : a -> NamedVect a n ss -> NamedVect a (S n) (s :: ss)
13
14 using (n : Nat, ss : Vect n String)
15   data Elem : Vect n String -> String -> Type where
16     Here : so (s == s') -> Elem {n=S n} (s :: ss) s'
17     There : Elem {n=n} ss s' -> Elem {n=S n} (s::ss) s'
18
19 elemCase : Elem {n=S n} (s :: ss) s' -> (so (s == s') -> a) -> (Elem ss s' -> a) -> a
20 elemCase (Here h) ifHere _ = ifHere h
21 elemCase (There th) _ ifThere = ifThere th
22
23 elemEmpty : Elem [] s -> a
24 elemEmpty (Here _)  impossible
25 elemEmpty (There _) impossible
26
27 -- | Decide whether a name is an element of a Vect of Strings.
28 decElem : (ss : Vect n String) -> (s : String) -> Dec (Elem ss s)
29 decElem [] s = No elemEmpty
30 decElem (s :: ss) s' with (choose (s == s'), decElem ss s')
31   decElem (s :: ss) s  | (Left h, _)   = Yes (Here h)
32   decElem (s :: ss) s' | (_, Yes rest) = Yes (There rest)
33   decElem (s :: ss) s' | (Right notHere, No notThere) =
34     No $ \h => elemCase h (\h' => soNot h' notHere) notThere
35
36 -- | Extract a named element through straightforward recursion on a proof of membership
```

```
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 []      prf          = elemEmpty prf

-- | Find a proof of membership, then extract the element
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

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

-- | Name an unnamed vector
applyNames : (ss : Vect n String) -> Vect n a -> NamedVect a n ss
applyNames [] [] = []
applyNames (s::ss) (x::xs)= x :: applyNames ss xs

-- | Convert to an unnamed vector
forgetNames : NamedVect a n ss -> Vect n a
forgetNames [] = []
forgetNames (x :: xs) = x :: forgetNames xs

-- | Convert to a list
toList : NamedVect a n ss -> List a
toList = Vect.toList . forgetNames
```

Providers.CSV

```
module Providers.CSV

import Data.NamedVect
import Decidable.Equality

import Providers
%language TypeProviders

%default total

-- | Get the lines of a file as a list of strings
partial
readLines : String -> IO (List String)
readLines fname = map (Strings.split (\c => List.elem c ['\n', '\r'])) (readFile fname)

-- | Split into columns
cols : Char -> String -> List String
cols delim row = map trim (split (==delim) row)

-- | Convert a List to a Vect n iff the list has n elements
lengthIs : (n : Nat) -> List a -> Maybe (Vect n a)
lengthIs Z     []        = Just []
lengthIs Z     (x :: xs) = Nothing
lengthIs (S n) []        = Nothing
```

```
25  lengthIs (S n) (x :: xs) = map (Vect.(::) x) (lengthIs n xs)
26
27  -- | A representation of the "schema" of a CSV file.
28  data CSVType : Type where
29    MkCSVType : (delim : Char) ->
30                (n : Nat) ->
31                (header : Vect n String) ->
32                CSVType
33
34  delim : CSVType -> Char
35  delim (MkCSVType d _ _) = d
36
37  colCount : CSVType -> Nat
38  colCount (MkCSVType _ n _) = n
39
40  header : (t : CSVType) -> Vect (colCount t) String
41  header (MkCSVType _ _ h) = h
42
43  vectEq : Eq a => Vect n a -> Vect n a -> Bool
44  vectEq Nil Nil = True
45  vectEq (x :: xs) (y :: ys) = x == y && vectEq xs ys
46
47  instance Eq CSVType where
48    (==) (MkCSVType delim1 n1 header1) (MkCSVType delim2 n2 header2) with (decEq n1 n2)
49      (==) (MkCSVType delim1 n header1)  (MkCSVType delim2 n header2)  | Yes refl =
50        delim1 == delim2 && vectEq header1 header2
51      (==) (MkCSVType delim1 n1 header1) (MkCSVType delim2 n2 header2) | No _ = False
52
53
54  -- A row corresponding to a schema is just a named vector matching the schema
55  Row : CSVType -> Type
56  Row (MkCSVType d n h) = NamedVect String n h
57
58  row_lemma : (t : CSVType) -> NamedVect String (colCount t) (header t) -> Row t
59  row_lemma (MkCSVType d n h) v = v
60
61
62  -- Attempt to read a string as a row in some schema
63  readRow : (t : CSVType) -> String -> Maybe (Row t)
64  readRow t r = map (row_lemma t . applyNames (header t)) .
65                lengthIs (colCount t) .
66                cols (delim t) $ r
67
68  readRows : (type : CSVType) -> List String ->
69             List (Maybe (Row type))
70  readRows type rows = map (readRow type) rows
71
72  -- Construct a CSV type from the first line of the CSV file
73  inferCSVType : (delim : Char) -> (header : String) -> CSVType
74  inferCSVType delim header =
75    let cs = cols delim header
76    in MkCSVType delim (length cs) (fromList cs)
77
78  %assert_total -- seems to be a totality checker bug: is possibly not total due to:
        {Prelude.Strings.unpack1010}
79  readCSV : (delim : Char) ->
80            (header : String) ->
81            List String ->
82            List (Maybe (NamedVect String (length (cols delim header)) (fromList (cols delim
```

```
                header))))
readCSV delim header rows = readRows (inferCSVType delim header) rows

-- Type provider for CSV. Attempt to infer a CSV type from the provided file.
partial
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

-- Read the well-formed rows from a CSV file according to some schema
partial
readCSVFile : (t : CSVType) -> String -> IO (Maybe (List (Row t)))
readCSVFile t file =
  do lines <- readLines file
     case lines of
       [] => return Nothing
       (h::body) =>
         if inferCSVType (delim t) h == t
           then return (Just (mapMaybe (readRow t) (drop 1 lines)))
           else return Nothing

test : NamedVect Int 3 ["a", "b", "c"]
test = applyNames _ [1,2,3]
```

Appendix C

Database Interaction Type Provider

Main

```
module Main

import Providers
import Providers.DB
import Providers.DBProvider

%language TypeProviders

%provide (testDB : Database) with loadSchema "test.sqlite"

query : Query testDB ["name":::TEXT, "wheels":::NULLABLE INTEGER, "description":::TEXT]
query = Project (Select (Product (T testDB  "people")
                                 (Rename (T testDB "transport") "id" "transport_id"))
                        (Col "id" INTEGER == Col "owner" INTEGER))
                ["name":::TEXT, "wheels":::NULLABLE INTEGER, "description":::TEXT]

main : IO ()
main = do q <- doQuery "test.sqlite" query
          putStrLn $ printTable q
```

Providers.DB

```
module Providers.DB

import Providers

import Data.BoundedList
import Decidable.Equality
import Language.Reflection
import Language.Reflection.Utils
import SQLiteConstants
import SQLiteTypes

%default total

```

```
14  varchar : (s : String) -> BoundedList Char n
15  varchar {n=n} s = take n (unpack s)
16
17  partial
18  getProofYes : Dec p -> p
19  getProofYes (Yes prf) = prf
20
21  partial
22  getProofNo : Dec p -> p -> _|_
23  getProofNo (No prf) = prf
24
25  isYes : Dec p -> Type
26  isYes (Yes _) = ()
27  isYes (No _) = _|_
28
29  isNo : Dec p -> Type
30  isNo (Yes _) = _|_
31  isNo (No _)  = ()
32
33  soNot : so p -> so (not p) -> _|_
34  soNot oh oh impossible
35
36  instance Cast Nat Int where
37    cast Z = 0
38    cast (S n) = 1 + (cast n)
39
40  namespace Types
41    data SQLType = INTEGER | TEXT | NULLABLE SQLType | REAL | BOOLEAN
42
43    instance Show SQLType where
44      show INTEGER = "INTEGER"
45      show TEXT = "TEXT"
46      show (NULLABLE t) = "NULLABLE " ++ show t
47      show REAL = "REAL"
48      show BOOLEAN = "BOOLEAN"
49
50    instance Eq SQLType where
51      INTEGER       == INTEGER        = True
52      TEXT          == TEXT           = True
53      (NULLABLE t1) == (NULLABLE t2)  = t1 == t2
54      REAL          == REAL           = True
55      BOOLEAN       == BOOLEAN        = True
56      a             == b              = False
57
58    interpSql : SQLType -> Type
59    interpSql INTEGER = Int
60    interpSql TEXT = String
61    interpSql (NULLABLE t) = Maybe (interpSql t)
62    interpSql REAL = Float
63    interpSql BOOLEAN = Bool
64
65    printVal : (interpSql t) -> String
66    printVal {t=INTEGER} x = show x
67    printVal {t=TEXT} x = x
68    printVal {t=NULLABLE t'} Nothing = "null"
69    printVal {t=NULLABLE t'} (Just x) = printVal x
70    printVal {t=REAL} x = show x
71    printVal {t=BOOLEAN} x = show x
72
```

```
73    intNotText : INTEGER = TEXT -> _|_
74    intNotText refl impossible
75
76    intNotNullable : INTEGER = NULLABLE t -> _|_
77    intNotNullable refl impossible
78
79    intNotReal : INTEGER = REAL -> _|_
80    intNotReal refl impossible
81
82    intNotBool : INTEGER = BOOLEAN -> _|_
83    intNotBool refl impossible
84
85    textNotNullable : TEXT = NULLABLE t -> _|_
86    textNotNullable refl impossible
87
88    textNotReal : TEXT = REAL -> _|_
89    textNotReal refl impossible
90
91    textNotBool : TEXT = BOOLEAN -> _|_
92    textNotBool refl impossible
93
94    nullableNotReal : NULLABLE t = REAL -> _|_
95    nullableNotReal refl impossible
96
97    nullableNotBool : NULLABLE t = BOOLEAN -> _|_
98    nullableNotBool refl impossible
99
100   realNotBool : REAL = BOOLEAN -> _|_
101   realNotBool refl impossible
102
103   nullableInjective : NULLABLE t = NULLABLE t' -> t = t'
104   nullableInjective refl = refl
105
106   instance DecEq SQLType where
107     decEq INTEGER INTEGER = Yes refl
108     decEq INTEGER TEXT = No intNotText
109     decEq INTEGER (NULLABLE t) = No intNotNullable
110     decEq INTEGER REAL = No intNotReal
111     decEq INTEGER BOOLEAN = No intNotBool
112     decEq TEXT INTEGER = No $ \h => intNotText (sym h)
113     decEq TEXT TEXT = Yes refl
114     decEq TEXT (NULLABLE t) = No textNotNullable
115     decEq TEXT REAL = No textNotReal
116     decEq TEXT BOOLEAN = No textNotBool
117     decEq (NULLABLE t) INTEGER = No $ \h => intNotNullable (sym h)
118     decEq (NULLABLE t) TEXT = No $ \h => textNotNullable (sym h)
119     decEq (NULLABLE t) (NULLABLE t') with (decEq t t')
120       decEq (NULLABLE t) (NULLABLE t)  | Yes refl = Yes refl
121       decEq (NULLABLE t) (NULLABLE t') | No p = No $ \h => p (nullableInjective h)
122     decEq (NULLABLE t) REAL = No nullableNotReal
123     decEq (NULLABLE t) BOOLEAN = No nullableNotBool
124     decEq REAL INTEGER = No $ \h => intNotReal (sym h)
125     decEq REAL TEXT = No $ \h => textNotReal (sym h)
126     decEq REAL (NULLABLE t) = No $ \h => nullableNotReal (sym h)
127     decEq REAL REAL = Yes refl
128     decEq REAL BOOLEAN = No realNotBool
129     decEq BOOLEAN INTEGER = No $ \h => intNotBool (sym h)
130     decEq BOOLEAN TEXT = No $ \h => textNotBool (sym h)
131     decEq BOOLEAN (NULLABLE t) = No $ \h => nullableNotBool (sym h)
```

```
132     decEq BOOLEAN REAL = No $ \h => realNotBool (sym h)
133     decEq BOOLEAN BOOLEAN = Yes refl
134
135   -- Necessary because type class resolution and universes are hard to unite
136   infix 9 <==>
137
138   (<==>) : (interpSql t1) -> (interpSql t2) -> Bool
139   (<==>) {t1=t1}         {t2=t2}         a        b        with (decEq t1 t2, t1)
140     (<==>) {t1=INTEGER}    {t2=INTEGER}    a        b        | (Yes refl, INTEGER)    = a
          == b
141     (<==>) {t1=TEXT}       {t2=TEXT}       a        b        | (Yes refl, TEXT)       = a
          == b
142     (<==>) {t1=NULLABLE t} {t2=NULLABLE t} Nothing  Nothing  | (Yes refl, NULLABLE t) =
          True
143     (<==>) {t1=NULLABLE t} {t2=NULLABLE t} (Just a) (Just b) | (Yes refl, NULLABLE t) = a
          <==> b
144     (<==>) {t1=REAL}       {t2=REAL}       a        b        | (Yes refl, REAL)       = a
          == b
145     (<==>) {t1=BOOLEAN}    {t2=BOOLEAN}    a        b        | (Yes refl, BOOLEAN)    = a
          == b
146     (<==>) {t1=t1}         {t2=t2}         a        b        | (_,        _)          =
          False
147
148
149 namespace Schemas
150   data Attribute : Type where
151     (:::) : String -> SQLType -> Attribute
152
153   infix 8 :::
154
155   instance Show Attribute where
156     show (c:::t) = c ++ ":::" ++ show t
157
158   getName : Attribute -> String
159   getName (n ::: t) = n
160
161   getType : Attribute -> SQLType
162   getType (n ::: t) = t
163
164   data AttrEq : Attribute -> Attribute -> Type where
165     attrRefl : {colMatch : so (c1 == c2)} -> AttrEq (c1 ::: t) (c2 ::: t)
166
167   data Schema : Type where
168     Nil : Schema
169     (::) : (a : Attribute) -> (s : Schema) -> Schema
170
171   instance Cast Schema (List Attribute) where
172     cast [] = []
173     cast (a::as) = a :: cast as
174
175   instance Show Schema where
176     show s = show (cast {to=List Attribute} s)
177
178   data SchemaEq : Schema -> Schema -> Type where
179     Done : SchemaEq [] []
180     Attr : AttrEq a1 a2 -> SchemaEq s1 s2 -> SchemaEq (a1 :: s1) (a2 :: s2)
181
182
183   colNames : Schema -> List String
```

```
184  colNames [] = []
185  colNames (c ::: _ :: s) = c :: colNames s
186
187  data HasType : Schema -> String -> SQLType -> Type where
188    Here : so (col == col') -> HasType (col' ::: t :: s) col t
189    There : HasType s col t -> HasType (col' ::: t' :: s) col t
190
191  nilSchemaEmpty : HasType [] c t -> _|_
192  nilSchemaEmpty Here impossible
193  nilSchemaEmpty (There _) impossible
194
195  hasTypeInv : so (not (col == c)) -> (HasType s col t -> _|_) -> HasType (c ::: t' :: s)
       col t -> _|_
196  hasTypeInv notEq _        (Here eq)   = soNot eq notEq
197  hasTypeInv _     notThere (There prf) = notThere prf
198
199
200  decHasType : (s : Schema) -> (col : String) -> (t : SQLType) -> Dec (HasType s col t)
201  decHasType [] c t = No nilSchemaEmpty
202  decHasType (col':::t' :: s) col t with (choose (col == col'), decEq t t', decHasType s
       col t)
203    decHasType (col':::t :: s) col t | (Left prf,    Yes refl, _)       = Yes (Here prf)
204    decHasType (col':::t' :: s) col t | (_,           _,        Yes prf) = Yes (There prf)
205    decHasType (col':::t' :: s) col t | (Right notEqual, _,        No notPresent) = No $
         \h => hasTypeInv notEqual notPresent h
206
207  lookup : (s : Schema) -> (col : String) -> Maybe (t ** HasType s col t)
208  lookup [] _ = Nothing
209  lookup (c:::t :: s) c' =
210    case choose ( c' == c) of
211      Left eq => Just (t ** Here eq)
212      Right neq => case lookup s c' of
213                     Nothing => Nothing
214                     Just (Ex_intro t prf) => Just (t ** There prf)
215
216  -- Convenience function for eliminating type annotations on queries
217  partial lookup' : (s : Schema) -> (col : String) -> SQLType
218  lookup' (c:::t :: s) c' = if c == c' then t else lookup' s c'
219
220
221  -- Gives both a proof of sub-schema-ness and a means of performing the projection
222  SubSchema : (sub, super : Schema) -> Type
223  SubSchema [] s = ()
224  SubSchema (col:::t :: s') s = (HasType s col t, SubSchema s' s)
225
226  decSubSchema : (sub, super : Schema) -> Dec (SubSchema sub super)
227  decSubSchema [] s = Yes ()
228  decSubSchema (col:::t :: s') s with (decHasType s col t, decSubSchema s' s)
229    | (Yes p1, Yes p2) = Yes (p1, p2)
230    | (No p1, _)       = No $ \(h, _) => p1 h
231    | (_, No p2)       = No $ \(_, h) => p2 h
232
233
234
235  -- The schema of a cartesian product -- unsafe due to no overlapping column check
236  product : Schema -> Schema -> Schema
237  product [] s' = s'
238  product (attr :: s) s' = attr :: (product s s')
239
```

```
240   namespace Occurs
241     data Occurs : Schema -> String -> Type where
242       Here : so (c == c') -> Occurs (c:::t :: s) c'
243       There : Occurs s c' -> Occurs (c:::t :: s) c'
244
245     noOccursNil : Occurs [] c -> _|_
246     noOccursNil (Here p) impossible
247     noOccursNil (There occ) impossible
248
249     occursInv : (c, c': String) -> (s : Schema) -> so (not (c == c')) -> (Occurs s c' ->
            _|_) -> Occurs (c:::t :: s) c' -> _|_
250     occursInv c c' s notHere notThere (Here here) = soNot here notHere
251     occursInv c c' s notHere notThere (There there) = notThere there
252
253     decOccurs : (s : Schema) -> (col : String) -> Dec (Occurs s col)
254     decOccurs [] c' = No noOccursNil
255     decOccurs (c:::t :: s) c' with (choose (c==c'), decOccurs s c')
256       | (Left yep, _) = Yes (Here yep)
257       | (Right nope, Yes inRest) = Yes (There inRest)
258       | (Right nope, No notInRest) = No $ \h => occursInv _ _ _ nope notInRest h
259
260
261
262   data Disjoint : Schema -> Schema -> Type where
263     EmptyDisjoint : Disjoint [] s
264     ConsDisjoint  : Disjoint s1 s2 -> (Occurs s2 c -> _|_) -> Disjoint (c ::: t :: s1) s2
265
266
267   decDisjoint_lemma1 : Occurs s2 c -> Disjoint (c ::: t :: s1) s2 -> _|_
268   decDisjoint_lemma1 occ (ConsDisjoint rest notOcc) = notOcc occ
269
270   decDisjoint : (s1, s2 : Schema) -> Dec (Disjoint s1 s2)
271   decDisjoint [] s2 = Yes EmptyDisjoint
272   decDisjoint (c ::: t :: s1) s2 with (decOccurs s2 c, decDisjoint s1 s2)
273     | (Yes occ, _) = No $ \h => decDisjoint_lemma1 occ h
274     | (_, No restBad) = No $ \(ConsDisjoint rest _) => restBad rest
275     | (No notIn, Yes restOk) = Yes (ConsDisjoint restOk notIn)
276
277
278   rename : (s : Schema) -> (from : String) -> (to : String) ->
279            Occurs s from -> (Occurs s to -> _|_) ->
280            Schema
281   rename (f:::t :: s) from to (Here eq) _     = to:::t :: s
282   rename (c:::t :: s) from to (There rest) ok = c:::t :: rename s from to rest (\h => ok
          (There h))
283
284
285
286
287
288 namespace Data
289   namespace Row
290     data Row : Schema -> Type where
291       Nil : Row []
292       (::) : (interpSql t) -> Row s -> Row (col ::: t :: s)
293
294     (++) : Row s1 -> Row s2 -> Row (product s1 s2)
295     (++) []          r2 = r2
296     (++) (col :: r1) r2 = col :: (r1 ++ r2)
```

```
297
298     printRow : Row s -> String
299     printRow [] = "\n"
300     printRow (c :: cs) = printVal c ++ " | " ++ printRow cs
301
302     -- Get the element of the column named in the HasType proof
303     getCol : (c : String) -> HasType s c t -> Row s -> interpSql t
304     getCol c (Here eq)      (v::_)    = v
305     getCol c (There inRest) (_::rest) = getCol c inRest rest
306
307     -- Perform a project by following the "recipe" in the proof of SubSchema
308     project : Row s -> (s' : Schema) -> SubSchema s' s -> Row s'
309     project row []          ()            = []
310     project row (c:::t::s') prf = getCol c (fst prf) row :: project row s' (snd prf)
311
312     renameRow : Row s -> (from, to : String) ->
313                 (okFrom : Occurs s from) -> (okTo : Occurs s to -> _|_) ->
314                 Row (rename s from to okFrom okTo)
315     renameRow (v::vs) _ _ (Here p) _ =  v :: vs
316     renameRow (v::vs) from to (There p) okTo = v :: renameRow vs from to p (\h => okTo
            (There h))
317
318   namespace Table
319     data Table : Schema -> Type where
320       Nil : Table s
321       (::) : Row s -> Table s -> Table s
322
323     count : Table s -> Nat
324     count []        = 0
325     count (r :: rs) = S (count rs)
326
327     (++) : Table s -> Table s -> Table s
328     (++) [] t2 = t2
329     (++) (r::rs) t2 = r :: rs ++ t2
330
331     map : (Row s1 -> Row s2) -> Table s1 -> Table s2
332     map f [] = []
333     map f (r :: rs) = f r :: map f rs
334
335     filter : (Row s -> Bool) -> Table s -> Table s
336     filter p [] = []
337     filter p (r :: rs) = if p r then r :: filter p rs else filter p rs
338
339     (*) : Table s1 -> Table s2 -> Table (product s1 s2)
340     (*) [] t2 = []
341     (*) (r1::r1s) t2 = (map (r1 ++) t2) ++ r1s * t2
342
343     printTable : Table s -> String
344     printTable {s=s} t = printHeader s ++ printTable' t
345     where printAttr : Attribute -> String
346           printAttr (col ::: tp) = col
347           printHeader : Schema -> String
348           printHeader [] = "\n"
349           printHeader (attr :: s') = printAttr attr ++ " | " ++ printHeader s'
350           printTable' : Table s -> String
351           printTable' [] = "\n"
352           printTable' (r :: rs) = printRow r ++ printTable' rs
353
354 namespace Expr
```

```
355
356   data Expr : Schema -> SQLType -> Type where
357     Col : (c : String) -> (t : SQLType) ->
358           {default tactics { compute; applyTactic findHasType; solve;}
359            what : HasType s c t} ->
360           Expr s t
361     (+) : Expr s INTEGER -> Expr s INTEGER -> Expr s INTEGER
362     (-) : Expr s INTEGER -> Expr s INTEGER -> Expr s INTEGER
363     (==) : Expr s t1 -> Expr s t2 -> Expr s BOOLEAN
364     NotNull : Expr s (NULLABLE t) -> Expr s BOOLEAN
365     Length : Expr s TEXT -> Expr s INTEGER
366     Const : (t : SQLType) -> (c : interpSql t) -> Expr s t
367
368   expr : Expr s t -> Row s -> (interpSql t)
369   expr (Col c t {what=what}) r = getCol c what r
370   expr (e1 + e2) r = expr e1 r + expr e2 r
371   expr (e1 - e2) r = expr e1 r - expr e2 r
372   expr (e1 == e2) r = expr e1 r <==> expr e2 r
373   expr (NotNull e) r = case expr e r of
374                          Nothing => False
375                          Just _  => True
376   expr (Length e) r = cast {to=Int} (length (expr e r))
377   expr (Const t c) r = c
378
379   %assert_total
380   compileExpr : Expr s t -> String
381   compileExpr (Col c t) = "\"" ++ c ++ "\""
382   compileExpr (e1 + e2) = "(" ++ compileExpr e1 ++ ") + (" ++ compileExpr e2 ++ ")"
383   compileExpr (e1 - e2) = "(" ++ compileExpr e1 ++ ") - (" ++ compileExpr e2 ++ ")"
384   compileExpr (e1 == e2) = "(" ++ compileExpr e1 ++ ") = (" ++ compileExpr e2 ++ ")"
385   compileExpr (NotNull e) = "(" ++ compileExpr e ++ ") is not null"
386   compileExpr (Length e) = "length(" ++ compileExpr e ++ ")"
387   compileExpr (Const TEXT str) = "'" ++ str ++ "'" -- FIXME escape strings
388   compileExpr (Const INTEGER i) = show i
389   compileExpr (Const (NULLABLE t) Nothing) = "null"
390   compileExpr {s=s} (Const (NULLABLE t) (Just x)) = compileExpr {s=s} (Const t x)
391   compileExpr (Const REAL r) = show r
392   compileExpr (Const BOOLEAN True) = "true"
393   compileExpr (Const BOOLEAN False) = "false"
394
395 namespace Database
396   data Database : Type where
397     Nil : Database
398     (::) : (String, Schema) -> Database -> Database
399
400   namespace HasTable
401     data HasTable : Database -> String -> Schema -> Type where
402       Here : so (n == n') -> HasTable ((n,s)::db) n' s
403       There : HasTable db n s -> HasTable ((n',s')::db) n s
404
405     getSchema : String -> Database -> Maybe Schema
406     getSchema tbl [] = Nothing
407     getSchema tbl ((name, schema)::tables) = if tbl == name then Just schema else
          getSchema tbl tables
408
409
410 namespace Automation
411
412   -- "Blunderbuss" tactic for taking care of very trivial goals
```

```
413   easy : List (TTName, Binder TT) -> TT -> Tactic
414   easy ctxt goal = try $ findAssumption ctxt goal ++
415                          [ (Refine (MN 0 "__II")) -- unit
416                          , Trivial                -- refl from context
417                          ]
418     where findAssumption : List (TTName, Binder TT) -> TT -> List Tactic
419           findAssumption [] _ = []
420           findAssumption ((n, b)::ctxt) goal =
421               if (goal == binderTy b) then [Refine n] else findAssumption ctxt goal
422
423
424   extractAttr : (TT, List TT) -> (TT, TT) -- Extract the components of attribute (c ::: t)
425   extractAttr (P _ (NS (UN ":::") ["Schemas", "DB", "Providers"]) _, [col, ty]) = (col, ty)
426   extractAttr (fn, args) = (P Ref (UN (show fn ++ "  " ++ show args)) Erased, Erased)
427
428   extractPair : (TT, List TT) -> (TT, TT) -- Extract from anything pairlike
429   extractPair (_, [a, b]) = (a, b)
430   extractPair (op, args) = (P Ref (UN (show op ++ "  " ++ show args)) Erased, Erased)
431
432   typeHere : TTName
433   typeHere = (NS "Here" ["Schemas", "DB", "Providers"])
434
435   typeThere : TTName
436   typeThere = (NS "There" ["Schemas", "DB", "Providers"])
437
438
439   findHasTypeProof : TT -> TT
440   findHasTypeProof goal =
441     case unApply goal of
442       (hasType, [s, c, t]) => hasTypeProof (unApply s) c t
443       _ => (P Ref "Wrong type" Erased)
444     where %assert_total hasTypeProof : (TT, List TT) -> TT -> TT -> TT
445           hasTypeProof (P _ (NS (UN "::") ["Schemas", "DB", "Providers"]) _, [hd, tl]) c t
                  =
446             let (c', t') = extractAttr (unApply hd) in
447             if (c == c' && t == t')
448               then mkApp (P Ref typeHere Erased) [c, c', t, tl, P Ref "oh" Erased]
449               else mkApp (P Ref typeThere Erased) [tl, c, t, c', t', hasTypeProof (unApply
                      tl) c t]
450           hasTypeProof fail c t = P Ref ("hasTypeProof failure for " ++ show c ++ " and "
                   ++ show t) Erased
451
452   findHasType : List (TTName, Binder TT) -> TT -> Tactic
453   findHasType ctxt goal = Exact $ findHasTypeProof goal
454
455
456   %assert_total
457   findSubSchemaProof : TT -> TT
458   findSubSchemaProof goal =
459     case unApply goal of
460       (P _ (MN _ "__Unit") _, []) => (P Ref (MN 0 "__II") Erased)
461       (n, [hasType, next]) => mkApp (P Ref "mkPair" Erased) [ hasType
462                                                             , next
463                                                             , findHasTypeProof hasType
464                                                             , findSubSchemaProof next
465                                                             ]
466       other => P Ref ("Fail " ++ show other) Erased
467
468   findSubSchema : List (TTName, Binder TT) -> TT -> Tactic
```

```
469   findSubSchema ctxt goal = Exact $ findSubSchemaProof goal
470
471   occursHere : TTName
472   occursHere = (NS "Here" ["Occurs", "Schemas", "DB", "Providers"])
473
474   occursThere : TTName
475   occursThere = (NS "There" ["Occurs", "Schemas", "DB", "Providers"])
476
477   -- Construct a proof term witnessing that some key is found in a schema
478   partial
479   findOccursProof : TT -> TT
480   findOccursProof goal =
481     case unApply goal of
482       (occurs, [s, c]) => occursProof (unApply s) c
483     where %assert_total occursProof : (TT, List TT) -> TT -> TT
484           occursProof (P _ (NS (UN "::") ["Schemas", "DB", "Providers"]) _, [hd, tl]) c =
485             let (col, t) = extractAttr (unApply hd) in
486             if col == c
487               then mkApp (P Ref occursHere Erased) [col, c, t, tl, P Ref (UN "oh") Erased]
488               else mkApp (P Ref occursThere Erased) [tl, c, col, t, occursProof (unApply
                      tl) c]
489           occursProof other c = (P Ref ("Failed to construct proof that " ++ show c ++ "
                  is in " ++ show other) Erased)
490
491   partial
492   findOccurs : List (TTName, Binder TT) -> TT -> Tactic
493   findOccurs ctxt goal = Exact $ findOccursProof goal
494
495   -- Given a premise that an item occurs in a context where it doesn't, give a proof that
          it doesn't
496   partial
497   findNotOccursProof : TT -> Maybe TT
498   findNotOccursProof h =
499     case unApply h of
500       (occurs, [s, c]) => notOccursProof' (unApply s) c
501       _ => Nothing
502     where
503           %assert_total notOccursProof' : (TT, List TT) -> TT -> Maybe TT
504           notOccursProof' (P _ (NS (UN "Nil") ["Schemas", "DB", "Providers"]) _,  []) c =
505             Just $ mkApp (P Ref (NS (UN "noOccursNil") ["Occurs", "Schemas", "DB",
                      "Providers"]) Erased) [c]
506           notOccursProof' (P _ (NS (UN "::") ["Schemas", "DB", "Providers"]) _, [hd, tl])
                  c =
507             let step = \tm => mkApp (P Ref (NS (UN "occursInv") ["Occurs", "Schemas",
                      "DB", "Providers"]) Erased)
508                                     [ snd (extractAttr (unApply hd))
509                                     , fst (extractAttr (unApply hd))
510                                     , c
511                                     , tl
512                                     , P Ref (UN "oh") Erased
513                                     , tm
514                                     ]
515             in [| step (notOccursProof' (unApply tl) c) |]
516           notOccursProof' found c =
517             Just (P Ref ("b: " ++ show (fst found) ++ " and " ++ show (snd found)) Erased)
518
519   -- Find a proof for a contradiction of an Occurs
520   findNotOccurs : List (TTName, Binder TT) -> TT -> Tactic
521   findNotOccurs ctxt (Bind n (Pi h) false) =
```

```
522         case findNotOccursProof h of
523           Just prf => Exact prf
524           Nothing => Refine ("Couldn't find proof for " ++ show h) -- easy ctxt (Bind n (Pi
                h) false)
525     findNotOccurs ctxt goal = Refine ("Goal not a negation of occurs: " ++ show goal)-- easy
          ctxt goal
526
527     hasTableHere : TTName
528     hasTableHere = (NS "Here" ["HasTable", "Database", "DB", "Providers"])
529
530     hasTableThere : TTName
531     hasTableThere = (NS "There" ["HasTable", "Database", "DB", "Providers"])
532
533     partial
534     findHasTableProof : TT -> TT
535     findHasTableProof goal =
536       case unApply goal of
537         (hasTable, [db, name, schema]) => hasTableProof (unApply db) name schema
538       where %assert_total hasTableProof : (TT, List TT) -> TT -> TT -> TT
539 --          hasTableProof a b c = (P Ref (show a ++ "    --        " ++  show b) Erased)
540           hasTableProof (P _ (NS (UN "::") ["Database", "DB", "Providers"]) _, [hd, tl])
                name schema =
541             let (name', schema') = extractPair (unApply hd) in
542             if name' == name
543               then mkApp (P Ref hasTableHere Erased) [name, name', schema, tl, P Ref (UN
                    "oh") Erased]
544               else mkApp (P Ref hasTableThere Erased) [tl, name, schema, name', schema',
                    tl, hasTableProof (unApply tl) name schema]
545           hasTableProof db name schema =  (P Ref ("Failed to construct proof that " ++
                show name ++ " is a table in " ++ show db ++ " with schema " ++ show
                schema) Erased)
546
547     partial
548     findHasTable : Nat -> List (TTName, Binder TT) -> TT -> Tactic
549     findHasTable _ ctxt goal = Exact $ findHasTableProof goal
550
551     findHasTable Z ctxt goal = seq [ Refine (NS (UN "Here") ["HasTable"]), Solve, Refine (UN
          "oh"), Solve]
552     findHasTable (S n) ctxt goal = Try (seq [ Refine (NS (UN "Here") ["HasTable"])
553                                             , Refine (UN "oh")
554                                             , Solve
555                                             ])
556                                        (seq [ Refine (NS (UN "There") ["HasTable"])
557                                             , Solve
558                                             , findHasTable n ctxt goal
559                                             ])
560
561
562
563 namespace Query
564
565     data Query : Database -> Schema -> Type where
566       T : (db : Database) -> (tbl : String) ->
567 --        {default tactics {compute ; applyTactic findHasTable 50 ;  }
568 --         ok : HasTable db tbl s} ->
569           {auto ok : getSchema tbl db = Just s} ->
570           Query db s
571       Union : Query db s -> Query db s -> Query db s
572       Product : Query db s1 -> Query db s2 ->
```

```
573               {default ()
574                ok : isYes (decDisjoint s1 s2)} ->
575               Query db (product s1 s2)
576     Project : Query db s -> (s' : Schema) ->
577               {default tactics { compute; applyTactic findSubSchema; solve; }
578                ok : SubSchema s' s } ->
579               Query db s'
580     Select : Query db s -> Expr s BOOLEAN -> Query db s
581     Rename : Query db s -> (from, to : String) ->
582              {default tactics { compute; applyTactic findOccurs; solve; }
583               fromOK : Occurs s from} ->
584              {default tactics { compute; applyTactic findNotOccurs; solve; }
585               toOK : Occurs s to -> _|_} ->
586              Query db (Schemas.rename s from to fromOK toOK)
587
588   getSchema : Query db s -> Schema
589   getSchema {s=s} _ = s
590
591   compileRename : (s : Schema) -> (from, to : String) -> String
592   compileRename s from to = foldl (++) "" (intersperse ", " (compileRename' s))
593     where
594       compileRename' : Schema -> List String
595       compileRename' [] = []
596       compileRename' (col:::t :: s) =
597         if col == from
598           then ("\"" ++ from ++ "\" AS \"" ++ to ++ "\"") :: colNames s
599           else col :: (compileRename' s)
600
601   compile : Query db s -> String
602   compile te = "SELECT * FROM (" ++ compile' te ++ ")"
603     where compile' : Query db s -> String
604           compile' (T db tbl) = tbl
605           compile' (Union t1 t2) = "(" ++ compile' t1 ++ ") UNION (" ++ compile' t2 ++ ")"
606           compile' (Product t1 t2) = "(" ++ compile' t1 ++ ") , (" ++ compile' t2 ++ ")"
607           compile' (Project t1 s') = "SELECT " ++
608                                      (foldl (++) ""
609                                        (intersperse ", "
610                                          (map (\c => "\"" ++ c ++ "\"") (colNames s')))) ++
611                                      " FROM (" ++ compile' t1 ++ ")"
612           compile' {s=s} (Select t e) = "SELECT " ++ (foldl (++) "" (intersperse ", " (map
                  (\c => "\"" ++ c ++ "\"") (colNames s)))) ++
613                                         " FROM (" ++ compile' t ++ ") WHERE " ++
                                                compileExpr e
614           compile' (Rename t from to) = "SELECT " ++ (compileRename (getSchema t) from to)
                  ++
615                                         " FROM (" ++ compile' t ++ ")"
616
617
618   partial  query : Query db s -> Table s
619 --  query (T t) = t
620   query (Union t1 t2) = (query t1) ++ (query t2)
621   query (Product t1 t2) = (query t1) * (query t2)
622   query (Project t s' {ok=ok}) = map (\r => project r s' ok) (query t)
623   query (Select t e) = filter (expr e) (query t)
624   query (Rename t from to {fromOK=fromOK} {toOK=toOK}) = map (\r => renameRow r from to
          fromOK toOK) (query t)
```

Providers.DBProvider

```
module Providers.DBProvider

import Providers.DB

import Providers.DBParser

%lib C "sqlite3"
%link C "sqlite_provider.o"
%include C "sqlite_provider.h"
%dynamic "libsqlite3"
%dynamic "./sqlite_provider.so"


namespace TypeProvider

  getTable : Ptr -> IO (Maybe (String, String))
  getTable ptr = do res <- mkForeign (FFun "sqlite3_step" [FPtr] FInt) ptr
                    if res /= SQLITE_ROW
                      then return Nothing
                      else do name <- mkForeign (FFun "sqlite3_column_text" [FPtr, FInt]
                           FString) ptr 0
                              sch <- mkForeign (FFun "sqlite3_column_text" [FPtr, FInt]
                                   FString) ptr 1
                              return (Just (name, sch))

  getTables : Ptr -> IO (List (String, String))
  getTables ptr = do tbl <- getTable ptr
                     case tbl of
                       Just x => do tbls <- getTables ptr
                                    return (the (List (String, String)) (x :: tbls))
                       Nothing => return List.Nil

  toDB : List (String, Schema) -> Database
  toDB [] = []
  toDB ((n, s) :: ts) = (n, s) :: (toDB ts)

  getDB : String -> IO Database
  getDB file = do ptr <- mkForeign (FFun "sqlite3_open_file" [FString] FPtr) file
                  stmt <- mkForeign (FFun "query_get_tables" [FPtr] FPtr) ptr
                  tblInfo <- getTables stmt
                  let asList = mapMaybe (getSchema . snd) tblInfo
                  return (toDB asList)

  loadSchema : String -> IO (Provider Database)
  loadSchema file = do db <- getDB file
                       return (Provide db)

  getResultVal : Ptr -> Int -> (t : SQLType) -> IO (interpSql t)
  getResultVal stmt i INTEGER = mkForeign (FFun "sqlite3_column_int" [FPtr, FInt] FInt)
       stmt i
  getResultVal stmt i TEXT = mkForeign (FFun "sqlite3_column_text" [FPtr, FInt] FString)
       stmt i
  getResultVal stmt i (NULLABLE t) = do tp <- mkForeign (FFun "sqlite3_column_type" [FPtr,
       FInt] FInt) stmt i
                                        if tp == SQLITE_NULL
                                          then return Nothing
```

```
52                                        else map Just (getResultVal stmt i t)
53    getResultVal stmt i REAL = mkForeign (FFun "sqlite3_column_double" [FPtr, FInt] FFloat)
          stmt i
54    getResultVal stmt i BOOLEAN = do val <- mkForeign (FFun "sqlite3_column_int" [FPtr,
          FInt] FFloat) stmt i
55                                     return (val /= 0)
56
57    getResultRow : Ptr -> Int -> (s : Schema) -> IO (Row s)
58    getResultRow stmt _ [] = return Row.Nil
59    getResultRow stmt i (c:::t :: s) = do val <- getResultVal stmt i t
60                                          rest <- getResultRow stmt (i+1) s
61                                          return (the (Row (c:::t :: s)) (val :: rest))
62
63    getResults : Ptr -> (s : Schema) -> IO (Table s)
64    getResults stmt s = do more <- mkForeign (FFun "sqlite3_step" [FPtr] FInt) stmt
65                           if more /= SQLITE_ROW
66                             then return Table.Nil
67                             else do r <- getResultRow stmt 0 s
68                                     next <- getResults stmt s
69                                     return (the (Table s) (r::next))
70
71    doQuery : String -> Query db s -> IO (Table s)
72    doQuery file q = do let q' = compile q ++ ";"
73                        db <- mkForeign (FFun "sqlite3_open_file" [FString] FPtr) file
74                        stmt <- mkForeign (FFun "prepare_query" [FPtr, FString] FPtr) db q'
75                        getResults stmt (getSchema q)
```

SQLiteConstants

```
1  module SQLiteConstants
2
3  -- Successful result
4  SQLITE_OK : Int
5  SQLITE_OK = 0
6
7  -- Error codes
8  -- SQL error or missing database
9  SQLITE_ERROR : Int
10 SQLITE_ERROR = 1
11
12 -- Internal logic error in SQLite
13 SQLITE_INTERNAL : Int
14 SQLITE_INTERNAL = 2
15
16 -- Access permission denied
17 SQLITE_PERM : Int
18 SQLITE_PERM = 3
19
20 -- Callback routine requested an abort
21 SQLITE_ABORT : Int
22 SQLITE_ABORT = 4
23
24 -- The database file is locked
25 SQLITE_BUSY : Int
26 SQLITE_BUSY = 5
27
28 -- A table in the database is locked
```

```
29  SQLITE_LOCKED : Int
30  SQLITE_LOCKED = 6
31
32  -- A malloc() failed
33  SQLITE_NOMEM : Int
34  SQLITE_NOMEM = 7
35
36  -- Attempt to write a readonly database
37  SQLITE_READONLY : Int
38  SQLITE_READONLY = 8
39
40  -- Operation terminated by sqlite3_interrupt()
41  SQLITE_INTERRUPT : Int
42  SQLITE_INTERRUPT = 9
43
44  -- Some kind of disk I/O error occurred
45  SQLITE_IOERR : Int
46  SQLITE_IOERR = 10
47
48  -- The database disk image is malformed
49  SQLITE_CORRUPT : Int
50  SQLITE_CORRUPT = 11
51
52  -- Unknown opcode in sqlite3_file_control()
53  SQLITE_NOTFOUND : Int
54  SQLITE_NOTFOUND = 12
55
56  -- Insertion failed because database is full
57  SQLITE_FULL : Int
58  SQLITE_FULL = 13
59
60  -- Unable to open the database file
61  SQLITE_CANTOPEN : Int
62  SQLITE_CANTOPEN = 14
63
64  -- Database lock protocol error
65  SQLITE_PROTOCOL : Int
66  SQLITE_PROTOCOL = 15
67
68  -- Database is empty
69  SQLITE_EMPTY : Int
70  SQLITE_EMPTY = 16
71
72  -- The database schema changed
73  SQLITE_SCHEMA : Int
74  SQLITE_SCHEMA = 17
75
76  -- String or BLOB exceeds size limit
77  SQLITE_TOOBIG : Int
78  SQLITE_TOOBIG = 18
79
80  -- Abort due to constraint violation
81  SQLITE_CONSTRAINT : Int
82  SQLITE_CONSTRAINT = 19
83
84  -- Data type mismatch
85  SQLITE_MISMATCH : Int
86  SQLITE_MISMATCH = 20
87
```

```
88  -- Library used incorrectly
89  SQLITE_MISUSE : Int
90  SQLITE_MISUSE = 21
91
92  -- Uses OS features not supported on host
93  SQLITE_NOLFS : Int
94  SQLITE_NOLFS = 22
95
96  -- Authorization denied
97  SQLITE_AUTH : Int
98  SQLITE_AUTH = 23
99
100 -- Auxiliary database format error
101 SQLITE_FORMAT : Int
102 SQLITE_FORMAT = 24
103
104 -- 2nd parameter to sqlite3_bind out of range
105 SQLITE_RANGE : Int
106 SQLITE_RANGE = 25
107
108 -- File opened that is not a database file
109 SQLITE_NOTADB : Int
110 SQLITE_NOTADB = 26
111
112 -- sqlite3_step() has another row ready
113 SQLITE_ROW : Int
114 SQLITE_ROW = 100
115
116 -- sqlite3_step() has finished executing
117 SQLITE_DONE : Int
118 SQLITE_DONE = 101
```

SQLiteTypes

```
1  module SQLiteTypes
2
3  SQLITE_INTEGER : Int
4  SQLITE_INTEGER = 1
5
6  SQLITE_FLOAT : Int
7  SQLITE_FLOAT = 2
8
9  SQLITE_BLOB : Int
10 SQLITE_BLOB = 4
11
12 SQLITE_TEXT : Int
13 SQLITE_TEXT = 3
14
15 SQLITE_NULL : Int
16 SQLITE_NULL = 5
```

sqlite_provider.h

```
1  #ifndef SQLITE_PROVIDER_H
2  #define SQLITE_PROVIDER_H
```

```
3
4  #include <stdio.h>
5  #include <sqlite3.h>
6
7  sqlite3 *sqlite3_open_file(char *filename);
8  unsigned char* sqlite3_get_schema(char* filename, char* table);
9  sqlite3_stmt *query_get_tables(sqlite3 *db);
10 sqlite3_stmt *prepare_query(sqlite3 *db, char* query);
11 #endif
```

sqlite_provider.c

```
1  #include "sqlite_provider.h"
2
3  sqlite3 *sqlite3_open_file(char *filename) {
4    sqlite3 *db;
5
6    int res = sqlite3_open(filename, &db);
7    if (res != SQLITE_OK) {
8      printf("Could not open db %s", filename);
9      return NULL;
10   }
11   else return db;
12 }
13
14 sqlite3_stmt *prepare_query(sqlite3 *db, char* query) {
15   sqlite3_stmt *ppStmt;
16   const char *rest;
17   int res = sqlite3_prepare_v2(db, query, strlen(query), &ppStmt, &rest);
18   if (res != SQLITE_OK) {
19     printf("Could not prepare query, res=%d\n\tquery=%s\n", res, query);
20     printf("Error: %s\n", sqlite3_errmsg(db));
21     return NULL;
22   }
23   return ppStmt;
24 }
25
26 sqlite3_stmt *query_get_tables(sqlite3 *db) {
27   sqlite3_stmt *ppStmt;
28   const char *rest;
29   int res = sqlite3_prepare_v2(db, "SELECT name, sql FROM sqlite_master WHERE
         type='table'", 100, &ppStmt, &rest);
30   if (res != SQLITE_OK) {
31     printf("Could not prepare query, res=%d\n", res);
32     return NULL;
33   }
34   return ppStmt;
35 }
36
37
38 unsigned char* sqlite3_get_schema(char *filename, char *table) {
39   const unsigned char *schema;
40
41   sqlite3 *db;
42
43   db = sqlite3_open_file(filename);
44
```

```
45    sqlite3_stmt *ppStmt;
46    const char *rest;
47    int res = sqlite3_prepare_v2(db, "SELECT sql FROM sqlite_master WHERE NAME = ?1", 100,
          &ppStmt, &rest);
48    if (res != SQLITE_OK) {
49      printf("Could not prepare statement, res=%d.\n", res);
50      return NULL;
51    }
52
53    res = sqlite3_bind_text(ppStmt, 1, table, -1, SQLITE_STATIC);
54    if (res != SQLITE_OK) {
55      printf("Could not bind table name %s, res=%d.\n", table, res);
56      return NULL;
57    }
58
59    res = sqlite3_step(ppStmt);
60    if (res != SQLITE_ROW) {
61      printf("No row, res=%d.\n", res);
62      return NULL;
63    }
64
65    schema = sqlite3_column_text(ppStmt, 0);
66    return schema;
67  }
```