A Functional Notation for Functional Dependencies

Matthias Neubauer [1] Peter Thiemann [2]

Institut für Informatik
Universität Freiburg,
Georges-Köhler-Allee Geb.079
D-79085 Freiburg i. Br., Germany

Martin Gasbichler [3] Michael Sperber [4]

Wilhelm-Schickard-Institut
Universität Tübingen
Sand 13
D-72076 Tübingen, Germany

Abstract

Functional dependencies help resolve many of the ambiguities that result from the use of multi-parameter type classes. They effectively enable writing programs at the type-level which significantly enhances the expressive power of Haskell's type system. Among the applications of this technique are the emulation of dependent types, and precise typechecking for XML and HTML combinator libraries. Unfortunately, the notation presently used for functional dependencies implies that the type-level programs are *logic* programs, but many of its applications are conceptually *functional* programs. We propose an alternative notation for functional dependencies which adds a functional-programming notation to Haskell's type classes and makes applications of functional dependencies significantly more readable. We apply the new notation to our examples and study the problems arising due to Haskell's open world assumption and overlapping instances.

Since the invention of type classes more than a decade ago [12], every new year has seen astonishing new applications and interesting extensions of the original idea. The most recent addition is Jones's incorporation of *functional dependencies* [8] which allow the formulation of tighter constraints on the instances

[1] Email: `neubauer@informatik.uni-freiburg.de`
[2] Email: `thiemann@informatik.uni-freiburg.de`
[3] Email: `gasbichl@informatik.uni-tuebingen.de`
[4] Email: `sperber@informatik.uni-tuebingen.de`

of multi-parameter type classes. A functional dependency is a declaration which specifies that a set of parameters of a type class uniquely determines another parameter. Such a specification avoids many ambiguous typings.

Functional dependencies constrain the instance declarations so that they specify a function at the type level. This functionality makes it feasible to write regular programs entirely evaluated by the type checker, opening the door for a wide range of potential applications [6,9].

Unfortunately, the use of functional dependencies is hampered by the traditional notation of type classes as relations and instances as logic programs computing these relations: As the resulting programs are getting more complex, they often become awkward and hard to read. This is not a fault of the mechanism of functional dependencies per se, but of the syntax offered by current implementations.

Thus, in this paper, we suggest a notational shift for type classes: View a type class as a type-level *function* instead of a type-level *relation*. We offer some realistic example applications of functional dependencies which would benefit significantly from such a notation. Moreover, we give a brief formal outline of how the new notation may be implemented by a translation to the traditional syntax. We present the relevant part of a larger example—type-safe combinator libraries for XML documents—and discuss some of the problems remaining.

1 Simulating Dependent Types

We start our investigation with two simple examples drawn from the realm of dependent types: formatted printing and specified list operations.

1.1 A type-safe `sprintf`

The `sprintf` function from the C-library is an example of an unsafe function that can be made type-safe by using a dependent type [1]: The first parameter of `sprintf` is a format specifier which determines the number and type of the remaining parameters. The idea here is that a type-level function computes the type of the remaining parameters from the format specifier. For this to work in Haskell, format specifiers cannot simply be strings with control characters but rather `String` constants or values from the following datatypes: [5]

```
data I f = I f
data C f = C f
data S f = S String f
```

Using these datatypes, the format specifier

```
S "Int: " (I (S ", Char: " (C ".")))
```

[5] Danvy [5] has shown that this particular example can be made to work relying only on ML-style polymorphism.

means "The literal string '`Int: `' followed by an integer followed by the literal string '`, Char: `' followed by a character and terminated by a period." A somewhat more convenient notation for the format specifier is the following:

```
S "Int: " $ I $ S ", Char: " $ C $ "."
```

The type of the format specifier contains an almost complete encoding of the format specifier itself, only omitting the string literals. In this case, the type is:

```
S (I (S (C String)))
```

A type class `SPRINTF` specifies a member function `sprintf1` which accepts a prefix, that it prepends to the output, a format specifier, and matching further arguments:

```
class SPRINTF m o | m -> o where
  sprintf1 :: String -> m -> o

instance SPRINTF String String where
  sprintf1 prefix str = prefix ++ str

instance SPRINTF a funa => SPRINTF (I a) (Int -> funa) where
  sprintf1 prefix (I a) i = sprintf1 (prefix ++ show i) a

instance SPRINTF a funa => SPRINTF (C a) (Char -> funa) where
  sprintf1 prefix (C a) c = sprintf1 (prefix ++ [c]) a

instance SPRINTF a funa => SPRINTF (S a) funa where
  sprintf1 prefix (S str a) = sprintf1 (prefix ++ str) a
```

The instance declarations comprise a logic program for computing `o` from `m` using the relation `SPRINTF`. In particular, the functional dependency `m -> o` is a kind of mode declaration.

The main entry point, `sprintf`, supplies an empty prefix to `sprintf1`:

```
sprintf :: SPRINTF m o => m -> o
sprintf = sprintf1 ""
```

For example, the type of

```
sprintIntChar = sprintf (S "Int: " $ I $ S ", Char: " $ C $ ".")
```

is

```
sprintIntChar :: Int -> Char -> String
```

When applied to an integer and a character, `sprintIntChar` yields

```
> sprintIntChar 42 'x'
"Int: 42, Char: x."
```

1.2 Specified list operations

Another classic example for the use of dependent types is length-indexed lists [13,9]. A translation of this example into Haskell first requires an encoding of natural numbers at the type level to encode list lengths:

```
data ZERO = ZERO
data SUCC n = SUCC n
```

`ADD` is a type class that encodes addition on these two datatypes at the type level using a functional dependency:

```
class ADD a b c | a b -> c
instance ADD ZERO b b
instance ADD a b c => ADD (SUCC a) b (SUCC c)
```

Again, the instance declarations for `ADD` constitute a small logic program.

Two datatypes encode heterogeneous lists on the value-level and on the type-level:

```
data NIL = NIL
data CONS x xs = CONS x xs
```

It is now possible to compute the length of a list statically via another logic program at the type level:

```
class LISTLENGTH xs len | xs -> len where
  listLength :: xs -> len

instance LISTLENGTH NIL ZERO where
  listLength NIL = ZERO
instance LISTLENGTH xs len
          => LISTLENGTH (CONS x xs) (SUCC len) where
  listLength (CONS x xs) = SUCC (listLength xs)
```

Here is a definition of an append operation on fixed-length lists:

```
class APPEND li1 li2 app | li1 li2 -> app where
  append :: li1 -> li2 -> app

instance APPEND NIL y y where
  append NIL y = y
instance APPEND xs ys app
          => APPEND (CONS x xs) ys (CONS x app) where
  append (CONS x xs) ys = CONS x (append xs ys)
```

With these declarations in place, it is natural to relate the definitions of `LISTLENGTH` and `ADD` to that of `APPEND`: Two concatenated lists must be as long as the sum of their lengths. The condition can be seen as a partial specification of `append`. Again, the constraint as a logic program is part of a revised instance declaration for `APPEND`:

```
instance (APPEND xs ys app,
          LISTLENGTH xs m,
```

```
          LISTLENGTH ys n,
          ADD m n s,
          LISTLENGTH app s)
           => APPEND (CONS x xs) ys (CONS x app) where
  append (CONS x xs) ys = CONS x (append xs ys)
```

Now, changing `LISTLENGTH` to something incompatible with the constraint results in a type error as soon as `append` occurs applied to a non-`NIL` list as first argument anywhere in a program. Consider the following buggy definition of `LISTLENGTH`:

```
instance LISTLENGTH NIL (SUCC ZERO) where
  listLength NIL = SUCC ZERO

instance LISTLENGTH xs len
           => LISTLENGTH (CONS x xs) (SUCC ZERO) where
  listLength (CONS x xs) = SUCC ZERO
```

As predicted, appending anything to `(CONS 1 NIL)` fails at compile time. Here is the error reported by Hugs:

```
> append (CONS 1 NIL) NIL
ERROR: Constraints are not consistent with functional dependency
*** Constraint       : ADD ZERO (SUCC ZERO) ZERO
*** And constraint   : ADD ZERO (SUCC ZERO) (SUCC ZERO)
*** For class        : ADD a b c
*** Break dependency : b a -> c
```

2 Functional is Better

The examples in the preceding section show that the logic-programming notation for type classes leads to poor readability in cases where the programmer really wants to define functions rather than predicates. In particular, we identify the following shortcomings:

- A functional language should express functional dependencies directly as functions. In the case of `LISTLENGTH`, where the `listLength` member mirrors the computation at the type level on the value level, the dichotomy is especially painful.
- The notation is occasionally difficult to read. Consider the final instance declaration of `APPEND`: The reader must discover that the type variable `s` is shared between the last argument of `ADD` and that of `LISTLENGTH`. In the `sprintf` example, nested applications are moved to the predicate set and linked via result variables.

In the following subsections, we propose a functional syntax for type-level functions and demonstrate their use with our examples.

2.1 Functional ***sprintf***

The modifications to the syntax are small and only affect the class and instance declarations. The syntax of expressions does not change. The header of the class declaration expresses the functional dependency directly. The first line of an instance declaration becomes a defining equation for a type function, the keyword `where` precedes the definition of the member function:

```
class SPRINTF :: m -> o where
  sprintf1 :: String -> m -> o

instance SPRINTF String = String                  where
  sprintf1 prefix str   = prefix ++ str

instance SPRINTF (I a)  = Int -> SPRINTF a        where
  sprintf1 prefix (I a) = \i  -> sprintf1 (prefix ++ show i) a

instance SPRINTF (C a)  = Char -> SPRINTF a       where
  sprintf1 prefix (C a) = \c   -> sprintf1 (prefix ++ [c]) a

instance SPRINTF (S a)      = SPRINTF a           where
  sprintf1 prefix (S str a) = sprintf1 (prefix ++ str) a

sprintf :: m -> SPRINTF m
sprintf = sprintf1 ""
```

The main conceptual change inherent in the notation is that type classes no longer specify predicates on types but rather functions. A class declaration by itself specifies the kind of the instance declarations; a new-style declaration

```
class SPRINTF :: m -> o
```

stands for the previous functional dependency

```
class SPRINTF m o | m -> o
```

The instance declarations define the corresponding function. This notation is arguably more natural than the logic-programming notation. This is not merely the case for this specific notation, but for a number of typical applications of functional dependencies.

With a functional notation on the type-level, qualified type schemes that incorporate predicates with functional dependencies can also be expressed more concisely: As the type annotation of `sprintf` shows, we can shift the application of `SPRINTF` to the position where the resulting type occurs.

2.2 List operations

The list operations also benefit from the new notation. This example also opens an additional issue, namely the question where to put the extra specification predicates that relate `ADD`, `LISTLENGTH`, and `APPEND`.

t ::=	a	type variable
|	C	type constructor
|	F	type function
|	$t\ t$	type application
d ::=	`class` $F :: a_1\ \ldots\ a_n \to a$	class declaration
|	`instance` $(t_1$ `==` $t'_1; \ldots; t_m$ `==` $t'_m) \Rightarrow F\ t_1\ \ldots\ t_n$ `=` t	instance declaration

Fig. 1. Syntax for functional instances

```
class LISTLENGTH :: xs -> len where
  listLength :: xs -> len
instance LISTLENGTH NIL = ZERO where
  listLength NIL = ZERO
instance LISTLENGTH (CONS x xs) = SUCC (LISTLENGTH xs) where
  listLength (CONS x xs) = SUCC (listLength xs)

class ADD :: a b -> c
instance ADD ZERO b = b
instance ADD (SUCC a) b = SUCC (Add a b)

class APPEND :: li1 li2 -> app where
  append :: li1 -> li2 -> app
instance APPEND NIL y = y where
  append NIL y = y
instance (LISTLENGTH (APPEND xs ys) ==
            ADD (LISTLENGTH xs) (LISTLENGTH ys))
           => APPEND (CONS x xs) ys = CONS x (APPEND xs ys) where
  append (CONS x xs) ys = CONS x (append xs ys)
```

This example shows that an instance declaration can carry an *axiom* consisting of an equality constraint: `LISTLENGTH (APPEND xs ys) == ADD (LISTLENGTH xs) (LISTLENGTH ys)`. The intended semantics of this constraint is unification. The original logic program models this by sharing the type variable `s`.

3 Functional Instances

We call the applicative notation for functional dependencies *functional instances*. The implementation of functional instances does not require any new machinery in the underlying Haskell system. The implementation is a syntactic translation of functional instances into standard Haskell instance declarations with functional dependencies. The translation resembles the flattening translation for functional logic programming languages [2]. This section gives an account of this translation. Figure 1 shows the syntax for functional

$$\vdash_\delta \texttt{class}\ F :: a_1\ \ldots\ a_n \to a \rightsquigarrow \texttt{class}\ F\ a_1\ \ldots\ a_n\ a \mid a_1\ \ldots\ a_n \to a$$

$$\frac{\vdash_\tau t \rightsquigarrow P_1 \mid t' \qquad \vdash_\sigma \{(l_1, r_1), \ldots, (l_m, r_m)\} \rightsquigarrow P_2}{\vdash_\delta \texttt{instance}\ (l_1\ \texttt{==}\ r_1; \ldots; l_m\ \texttt{==}\ r_m) \Rightarrow F\ t_1\ \ldots\ t_n\ \texttt{=}\ t}$$

$$\rightsquigarrow \texttt{instance}\ P_1, P_2 \Rightarrow F\ t_1\ \ldots\ t_n\ t'$$

$$\vdash_\tau a \rightsquigarrow \emptyset \mid a$$

$$\vdash_\tau C \rightsquigarrow \emptyset \mid C$$

$$\frac{\vdash_\tau t_1 \rightsquigarrow P_1 \mid t_1' \qquad \vdash_\tau t_2 \rightsquigarrow P_2 \mid t_2'}{\vdash_\tau t_1\ t_2 \rightsquigarrow P_1, P_2 \mid t_1'\ t_2'}$$

$$\frac{\vdash_\phi t \rightsquigarrow P \mid t'}{\vdash_\tau t \rightsquigarrow P, t'\ a \mid a}\quad a \text{ fresh}$$

$$\frac{\vdash_\phi t_1 \rightsquigarrow P_1 \mid t_1' \qquad \vdash_\tau t_2 \rightsquigarrow P_2 \mid t_2'}{\vdash_\phi t_1\ t_2 \rightsquigarrow P_1, P_2 \mid t_1'\ t_2'}$$

$$\vdash_\phi F \rightsquigarrow \emptyset \mid F$$

Fig. 2. Translation to old-style class and instance definitions

instances. An instance declaration may specify a set of axioms that must be fulfilled for the instance to apply. $\texttt{instance}\ F\ t_1\ \ldots\ t_n\ \texttt{=}\ t$ is an abbreviation for $\texttt{instance}\ () \Rightarrow F\ t_1\ \ldots\ t_n\ \texttt{=}\ t$.

In addition to the constraints imposed by the grammar, in every declaration $\texttt{instance}\ (\ldots) \Rightarrow F\ t_1\ \ldots\ t_n\ \texttt{=}\ t$, the terms $t_1, \ldots, t_n$ are *constructor terms*, meaning that no type function symbols F occur in them. Also, each type function of arity n must always be fully applied to n arguments.

The translation specified in Figures 2 and 3 is defined using five judgments:

(i) The judgment $\vdash_\delta d \rightsquigarrow d'$ maps a new-style definition, d, to an old-style definition, d'. It is used for translating class and instance definitions. The translation adds a result parameter to the class definition and states the functional dependency. For an instance definition, it translates the right-hand side of the definition into a predicate set and a type term.

(ii) The judgment, $\vdash_\tau t \rightsquigarrow P \mid t'$ translates a type term, t, into a predicate

$$\vdash_\sigma \emptyset \rightsquigarrow \emptyset$$

$$\frac{\vdash_\omega (l, r) \rightsquigarrow P_1 \qquad \vdash_\sigma \{x_2, \ldots, x_n\} \rightsquigarrow P_2}{\vdash_\sigma \{(l, r), x_2, \ldots, x_n\} \rightsquigarrow P_1, P_2}$$

$$\frac{\vdash_\tau t_1 \rightsquigarrow P_1 \mid t_1' \qquad \vdash_\tau t_2 \rightsquigarrow P_2 \mid t_2'}{\vdash_\omega (t_1, t_2) \rightsquigarrow P_1, P_2, \mathtt{EQV}\ t_1'\ t_2'}$$

Fig. 3. Translation of axioms

set, P, and a type, t'. Its main job is the translation of constructor terms, which is straightforward.

(iii) The translation of applications of type functions is left to the judgment, $\vdash_\phi t \rightsquigarrow P \mid t'$. It is only defined for terms t of the form $F\ t_1\ \ldots\ t_n$. All arguments, t_i, are translated using $\vdash_\tau\rightsquigarrow$ and the final rule that connects $\vdash_\phi\rightsquigarrow$ to $\vdash_\tau\rightsquigarrow$ adds the result parameter, a, as a fresh variable and moves the resulting predicate into the predicate set.

(iv) The judgment, $\vdash_\sigma \{(l_1, r_1), \ldots\} \rightsquigarrow P$, translates a set of equations, l_i `==` r_i, into a predicate, P. Its only function is to dispatch single equations to the next judgment.

(v) The judgment, $\vdash_\omega (t_1, t_2) \rightsquigarrow P$, translates a single equation into a predicate set, P. It translates t_1 and t_2 into predicates and type expressions without type functions. It enforces equality of t_1 and t_2 by making use of an auxiliary type class `EQV a b`. This class specifies that `a` and `b` must be equal:

```
class EQV a b | a -> b, b -> a
instance EQV a a
```

The functional dependencies in the class declaration specify that `EQV` is an invertible function. The only instance defines that `EQV` is the identity function.

It is a simple exercise to show that the judgment $\vdash_\delta\rightsquigarrow$ really defines a function.

Lemma 3.1 *If $\vdash_\delta d \rightsquigarrow d'$ and $\vdash_\delta d \rightsquigarrow d''$ then d' is equal to d'' up to renaming of free variables.*

4 A Larger Example: Regular Expressions

Besides giving a larger, real-life example, we also pinpoint new problems with pattern matching on types in this section.

In our work on generating valid HTML and XML documents in Haskell

[11], we model the restrictions that a DTD poses on the content of one particular XML element. The key ingredient in this model is a type class `AddTo elem subelem`, which relates each element to its corresponding admissible subelements. This approach requires that each element can be identified via its type.

A DTD specifies, in principle, a regular language of subelements for the contents of each element. It is straightforward to implement a simple recognizer for each element specified in the DTD as a regular expression matcher. Typechecking the uses of the element constructors then requires keeping track of the states of this matching automaton at the type level. The transition function is a generalization of the `AddTo` class, namely a class `NEXTSTATE subelem elem elem'`. In this approach, each element has an initial type (corresponding to the initial state of its automaton) and adding subelements changes the type of the element according to the transition function.

4.1 Data-Level Regular Expressions

Because of the laborious nature of standard type-level programs, we first present the transition function in the more familiar value-level syntax; this simplifies the presentation. First, here is a datatype for regular expressions:

```
type Symbol = String

data Regexp =
     Empty | Epsilon | Atom Symbol |
     Seq Regexp Regexp | Alt Regexp Regexp | Star Regexp
```

The transition function `nextState` maps an input symbol a and a regular expression r to a regular expression r' so that $L(r') = \{w|aw \in L(r)\}$. This is a well-known trick for matching a string to a regular expression. The same trick can also be used to translate a regular expression directly into a deterministic finite state automaton with regular expressions as its states [3]:

```
nextState :: Symbol -> Regexp -> Regexp
nextState a Empty       = Empty
nextState a Epsilon     = Empty
nextState a (Atom b)    = if a == b then Epsilon else Empty
nextState a (Seq r1 r2) = if finalState r1
                          then alt ra rb
                          else ra
                          where ra = seq (nextState a r1) r2
                                rb = nextState a r2
nextState a (Alt r1 r2) = alt (nextState a r1) (nextState a r2)
nextState a (Star r)    = seq (nextState a r) (Star r)
```

The `alt` and `seq` functions are “smart” versions of the corresponding `Alt` and `Seq` data constructors which use standard algebraic identities to simplify the resulting regular expressions on the fly; see Appendix A for their definitions.

The auxiliary `finalState` function (occasionally called `nullable` in the parsing folklore), when applied to a regular expression r, yields `True` when $\varepsilon \in L(r)$, thus identifying final states in the automaton:

```
finalState :: Regexp -> Bool
finalState Empty       = False
finalState Epsilon     = True
finalState (Atom b)    = False
finalState (Seq r1 r2) = finalState r1 && finalState r2
finalState (Alt r1 r2) = finalState r1 || finalState r2
finalState (Star r)    = True
```

4.2 Type-Level Regular Expressions

One possible approach to lifting the regular expression matcher to the type level is to explicitly compute the automaton first and translate it into a set of instance declarations. This is somewhat more involved than merely implementing the transition function. Consequently, it is more desirable to simply lift the entire computation of the automaton to the type level, preserving as much of the structure of the value-level program as possible.

The first step is to lift the declaration of regular expressions to the type level. This works just like the natural-number and fixed-length list examples above:

```
data EMPTY   = EMPTY
data EPSILON = EPSILON
data ATOM t  = ATOM t
data SEQ r s = SEQ r s
data ALT r s = ALT r s
data STAR r  = STAR r
```

In the context of the HTML combinator library, the argument type of `ATOM` is a tag such as `DL`, `DD`, or `DT`. Each of these tags is represented by a corresponding (singleton) type, like `EMPTY` or `EPSILON`.

The transition function maps a state and a symbol to the next state. Hence, it must be modeled using a type function `NEXTSTATE`: [6]

```
class NEXTSTATE :: t s -> s' where
  nextState :: t -> s -> s'
```

For readability, we omit some definitions of the `nextState` member function in the running text (see Appendix B for the complete definitions). The first two cases are simple:

```
instance NEXTSTATE t EMPTY = EMPTY where
  nextState t EMPTY = EMPTY
instance NEXTSTATE t EPSILON = EMPTY where
  nextState t EPSILON = EMPTY
```

[6] A version of the code written in the traditional notation for functional dependencies is in Appendix B.

But the next case, dealing with ATOM, raises an interesting problem: The conditional in the original program must distinguish between a matching and a non-matching input symbol. Hence, it must be able to compare types and "branch" on the results. The naive approach would be this one:

```
instance NEXTSTATE t (ATOM t) = EPSILON where ...
instance NEXTSTATE t (ATOM s) = EMPTY where ...
```

Unfortunately, the instances violate the functional dependency: all parameters are simultaneously unifiable by mapping s to t, but the results are different.

In a more expressive language, it would be possible to state that $t \neq s$ in the second instance. Unfortunately, it is not obvious how such constraints might be incorporated into the type language because inequations are not admissible predicates in Jones's framework of qualified types [7] (they are not closed under substitution). In addition, it is not clear if term equality is really the predicate that we want to test.

A working solution introduces an explicit equality test in the form of a type class EQUAL:

```
class EQUAL :: s t -> b where
  equal :: s -> t -> b
instance EQUAL DL DL = TRUE  where equal DL DL = TRUE
instance EQUAL DL DT = FALSE where equal DL DT = FALSE
instance EQUAL DL DD = FALSE where equal DL DD = FALSE
⋮
```

where TRUE and FALSE are the obvious singleton types. To be useful, there must be an instance definition for each pair of types, which are admissible symbol types. Subsection 4.4 discusses the problem in more detail.

With this in place, the ATOM case turns into a single instance:

```
instance NEXTSTATE t (ATOM s) = IF (Equal s t) EPSILON EMPTY where
  nextState t (ATOM s) = cond (equal s t) EPSILON EMPTY
```

where the IF type class implements a conditional on the level of types:

```
class IF :: b t1 t2 -> t where
  cond :: b -> t1 -> t2 -> t
instance IF TRUE  t1 t2 = t1 where cond TRUE t1 t2 = t1
instance IF FALSE t1 t2 = t2 where cond FALSE t1 t2 = t2
```

Interestingly, this conditional "evaluates" both branches, potentially even before the condition itself is known.

Sequences are the most difficult case. Fortunately, the techniques developed so far apply in the same way:

```
instance NEXTSTATE t (SEQ r1 r2) =
           IF (FINALSTATE r1)
              (ALT (SEQ (NEXTSTATE t r1) r2) (NEXTSTATE t r2))
              (SEQ (NEXTSTATE t r1) r2) where
```

```
  nextState t (SEQ r1 r2) =
    let ra = seq' (nextState t r1) r2
        rb = nextState t r2
    in  cond (finalState r1) ra (alt' ra rb)
```

The same holds for `ALT` and `STAR`:

```
instance NEXTSTATE t (ALT r1 r2) =
           ALT (NEXTSTATE t r1) (NEXTSTATE t r2) where
  nextState t (ALT r1 r2) =
    alt' (nextState t r1) (nextState t r2)
instance NEXTSTATE t (STAR r) =
           SEQ (NEXTSTATE t r) (STAR r) where
  nextState t (STAR r) =
    seq' (nextState r) (STAR r)
```

The definition of `FINALSTATE` is straightforward to derive from the value-level function `finalState`. It is omitted because it poses no additional problems. It relies on type-level versions of `and` and `or`, `AND` and `OR`, which are also straightforward.

4.3 Smart Constructors

In the data-level formulation, smart versions of `Alt` and `Seq` called `alt` and `seq` simplify the resulting regular expressions on-the-fly. When computing a matching automaton at the data level, this is strictly necessary to obtain a finite set of states. The type-level does not need to enumerate all the states, so it is possible to do without simplification. However, after several transitions, the type terms and their evidence-data terms grow enormously and slow down the type checker considerably.

Implementing smart constructors at the type level poses some interesting additional problems. The smart versions of `ALT` and `SEQ` are type classes called `ALT'` and `SEQ'`. Their class declarations are as follows:

```
class SEQ' :: r1 r2 -> r
class ALT' :: r1 r2 -> r
```

Their instance declarations can be easily derived from their data-level counterparts. However, they must be spelled out in considerably more detail. In particular, most variables must be eliminated from the patterns. For example:

```
instance ALT' EMPTY r     = r
instance ALT' r     EMPTY = r
```

These two instances overlap and hence the type checker rejects them. Even the implemented extensions for overlapping instances [10] do not help in this case because each instance matches the other.

One approach to make the instances non-overlapping is to define instances for all combinations of parameters `r1` and `r2` to `ALT'`. However, this results in a quadratic number of instances.

A better approach is to *sequentialize* the class ALT'. In this approach, ALT' is only responsible for dispatching on the first parameter. Dispatching on the second parameter is left to yet another type class ALT''. Sequentializing "only" doubles the number of instances. Here is the beginning of that effort:

```
instance ALT' EMPTY r2 = r2
instance ALT' r1    r2 = ALT'' r1 r2
```

The last line gives rise to an overlapping instance, but it can be resolved using the extension for overlapping instances.[7]

Ideally, an implementation of ALT'' would look as follows:

```
class ALT'' :: r1 r2 -> r
instance ALT'' r1 EMPTY = r1
instance ALT'' r1 r2    = ALT r1 r2
```

Unfortunately, the last line breaks the functional dependency: in the case where r2 = EMPTY, the last line says that ALT'' r1 EMPTY = ALT r1 EMPTY which seems to contradict the functional dependency of ALT'' (at least, this is what the implementations tell us). Consequently, all the alternatives need to be listed:

```
instance ALT'' r1 EPSILON     = ALT r1 EPSILON
instance ALT'' r1 (ATOM t)    = ALT r1 (ATOM t)
instance ALT'' r1 (SEQ s1 s2) = ALT r1 (SEQ s1 s2)
instance ALT'' r1 (ALT s1 s2) = ALT r1 (ALT s1 s2)
instance ALT'' r1 (STAR s)    = ALT r1 (STAR s)
```

4.4 The Problem with Equality

The astute reader will have asked why the type class EQUAL is defined as a function that explicitly maps a pair of types to TRUE or FALSE. It would be much more in line with typical type class programming to define a type class EQUAL2 along the following lines (in the traditional syntax):

```
class EQUAL2 a b
instance EQUAL2 DL DL
```

[7] Unfortunately, the February 2001 release of Hugs 98 is not able to correctly process the instances of ALT'. To obtain runnable code, it is necessary to expand the second line by giving the following set of instance definitions:

```
instance ALT' EMPTY r2 r2
instance (ALT'' EPSILON r2 r) => ALT' EPSILON r2 r
instance (ALT'' (ATOM t) r2 r) => ALT' (ATOM t) r2 r
instance (ALT'' (SEQ s1 s2) r2 r) => ALT' (SEQ s1 s2) r2 r
instance (ALT'' (ALT s1 s2) r2 r) => ALT' (ALT s1 s2) r2 r
instance (ALT'' (STAR r1) r2 r) => ALT' (STAR r1) r2 r
```

These instance definitions require a facility to turn off the termination guarantee of predicate rewriting (thus making type checking undecidable) because there are non-variable types in the predicates.

```
instance EQUAL2 DT DT
instance EQUAL2 DD DD
⋮
```

This type class could even be specified with a linear number of instance declarations.

Unfortunately, `EQUAL2` is not up to the job. Since the Haskell class system is designed with an open-world assumption in mind, it is impossible to extract negative information from the definition of `EQUAL2`. A predicate like `EQUAL2 DL DT` simply remains unreduced in the context. Hence, it cannot be used to select a particular instance.

Neither does the class `EQV` from Section 3 perform the desired task. Using the predicate `EQV t1 t2` immediately unifies `t1` and `t2`, thereby making it impossible to reach the `EMPTY` case for `NEXTSTATE t (ATOM s)`.

5 Towards better Type-Level Pattern Matching

While the syntax proposed in the previous sections is appealing and makes type-level programming much easier, it still leaves much to be desired. Most prominently, function definitions on the data level are still more readable than the ones on the type level, largely because data-level functions can utilize wild-card patterns and type-level functions cannot. Unfortunately, a naive translation of wild-card patterns to instances gives rise to overlapping instances. These overlapping instances are not easily resolved (witness Section 4.3), and give rise to subtle type checking problems.

Part of the problem lies in the as yet unexplored interaction of functional dependencies and overlapping instances. Since there is no underlying theory that encompasses both, there is still work to be done here.

Another part of the problem stems from the fact that type classes are the only structuring tool available for types. Data values can be classified by types in a much more rigorous fashion than types can be classified by type classes. This weak classification of types is due to the open nature of type classes: The typing of a program must not depend on the instances known at compile time. Instead, the program is guaranteed to work regardless of the instances that are added later on. Hence, it is impossible to automatically expand a wild-card pattern into the potential parameter types because this set might be different depending on the context of use of a particular module.

Moreover, the simplification example (Sec. 4.3), requires sequentializing the pattern matching. This "pattern matching by hand" seems wasteful, but is impossible to automate, as pattern-match compilation heavily relies on wild-card patterns.

A potential way out of the problem with wild-card patterns (and hence with pattern matching, too) lies in enriching the kind structure. Currently, the only kinds available are types, $*$, and arrow kinds, $\kappa \to \kappa'$. An *enumerated* *kind* defines a set of types, but in contrast to a type class, its set of types cannot be changed once it is defined.

The proposed notation for the simplification example (Sec. 4.3) is the following:

```
class ALT' r1 r2 -> r with
  r1, r2 <- { EMPTY, EPSILON, ATOM t, SEQ r1 r2, ALT r1 r2, STAR r1 }
```

Using this information, the compiler can compile pattern matching by expanding wild cards. [8]

6 Getting It Back

All examples presented in this paper exclusively rely on type classes defined using functional instances. The notation does not subsume the old notation for type classes. On the other hand, it is easy to embed old-style type classes into functional instances by always returning a dummy `Success` type from the type-level functions.

Moreover, the notation for functional instances also does not subsume functional dependencies. Specifically, functional instances do not not cover the following cases:

(i) functional dependencies which do not cover all of the variables of the type class

(ii) multiple functional dependencies in a single type class

In the first case, a class written using functional instances introduces type variables on the left-hand side which do not occur on the right-hand side of its instances. Naive implementations of type inference for functional dependencies might miss opportunities for improvement in this case. However, it is straightforward to detect unused left-hand-side type variables and omit them from the generated functional dependencies.

The second case is more involved, and it is hard to assess the value of multiple functional dependencies, as there are few published examples which make fundamental use of them. (Jones's original paper on functional dependencies [8] contains not a single concrete example for multiple dependencies, and the examples in McBride's collection [9] also work without them in concrete

---

[8] Curiously, it *is* already possible to define type classes that correspond to singleton kinds. For example, the class

```
class STrue true | -> true
```

has a functional dependency, which states that its parameter `true` depends on nothing. Hence, it must be a constant function (or undefined). Only a single instance declaration is possible:

```
instance STrue TRUE
```

Any use of `STrue a` in a context immediately simplifies to `a = TRUE`.

applications.) In any case, it seems it is usually possible to rewrite multiple functional dependencies into multiple type classes implementing the different modes specified by them.

7 Conclusions

We have shown examples for performing non-trivial computations on types using extensions of the Haskell class mechanism. We propose an alternative, applicative syntax for specifying type functions and we provide a translation from new-style classes and instances to the original formulation.

While we have shown how to translate the new functional instances into the old functional dependencies, we have not addressed the issue of how to do the reverse. Moreover, there is the question of how to define and handle ordinary type classes without functional dependencies in the new framework. One simple option is to keep the old notation entirely. Another option would be to reinterpret all type classes as functions rather than predicates. This means that inheritance constraints could also be formulated using equations over the values of type-class applications. The right mix is the subject of ongoing work.

Given the fragile nature of the available implementations of functional dependencies and the drawbacks listed in Section 5, we feel that the support for type-level programming is still in its infancy. To get the best of both worlds, there must be a better way of classifying types than type classes. The natural proposition here is to look to dependently typed systems and to allow lifting data-level computation to the type level [4]. This would solve the classification problem (since every lifted data value belongs to a lifted type, which is a kind), and address the problems about pattern matching and properties, too.

Also, since pattern matching has a well-defined semantics, such a step would alleviate the problems with overlapping instances. It remains, of course, to integrate this kind of computations into a type checker. This is also part of our ongoing work.

References

[1] Augustsson, L., *Cayenne—a language with dependent types*, in: P. Hudak, editor, *Proc. International Conference on Functional Programming 1998* (1998).

[2] Barbuti, R., M. Bellia, G. Levi and M. Martelli, *LEAF: A language which integrates logic, equations and functions*, in: D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, 1986 .

[3] Brzozowski, J. A., *Derivatives of regular expressions*, Journal of the ACM **11** (1964), pp. 481–494.

[4] Cardelli, L., *Phase distinctions in type theory* (1988), unpublished Manuscript.

[5] Danvy, O., *Functional unparsing*, Journal of Functional Programming **8** (1998), pp. 621–625.

[6] Hallgren, T., *Fun with functional dependencies*, in: *Joint Winter Meeting of the Departments of Science and Computer Engineering, Chalmers University of Technology and Göteborg University*, Varberg, Sweden, 2001, `http://www.cs.chalmers.se/~hallgren/Papers/wm01.html`.

[7] Jones, M. P., "Qualified Types: Theory and Practice," Cambridge University Press, Cambridge, UK, 1994.

[8] Jones, M. P., *Type classes with functional dependencies*, in: G. Smolka, editor, *Proc. 9th European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science (2000), pp. 230–244.

[9] McBride, C., *Faking it—simulating dependent types in Haskell* (2001), `http://www.dur.ac.uk/~dcs1ctm/faking.ps`.

[10] Peyton Jones, S., M. Jones and E. Meijer, *Type classes: An exploration of the design space*, in: J. Launchbury, editor, *Proc. of the Haskell Workshop*, Amsterdam, The Netherlands, 1997, yale University Research Report YALEU/DCS/RR-1075.

[11] Thiemann, P., *Modeling HTML in Haskell*, in: *Practical Aspects of Declarative Languages, Proceedings of the Second International Workshop, PADL'00*, number 1753 in Lecture Notes in Computer Science, Boston, Massachusetts, USA, 2000, pp. 263–277.

[12] Wadler, P. and S. Blott, *How to make ad-hoc polymorphism less ad-hoc*, in: *Proc. 16th Annual ACM Symposium on Principles of Programming Languages* (1989), pp. 60–76.

[13] Xi, H. and F. Pfenning, *Dependent types in practical programming*, in: A. Aiken, editor, *Proc. 26th Annual ACM Symposium on Principles of Programming Languages* (1999), pp. 214–227.

A Smart Constructors

These are the value-level definitions of the `seq` and `alt` smart constructors:

```
seq Empty r = Empty
seq r Empty = Empty
seq Epsilon r = r
seq r Epsilon = r
seq (Seq r1 r2) r3 = Seq r1 (seq r2 r3)
seq r1 r2 = Seq r1 r2
```

```
alt Empty r = r
alt r Empty = r
alt Epsilon Epsilon = Epsilon
alt Epsilon (Star r) = Star r
alt (Star r) Epsilon = Star r
alt (Alt r1 r2) r3 = Alt r1 (alt r2 r3)
alt r1 r2 = Alt r1 r2
```

B Instance Declarations for NEXTSTATE and FINALSTATE

Here is the code for the `NEXTSTATE` class in the traditional syntax for functional dependencies:

```
class NEXTSTATE t s s' | t s -> s' where
  nextState :: t -> s -> s'

instance NEXTSTATE t EMPTY EMPTY where
  nextState t EMPTY = EMPTY

instance NEXTSTATE t EPSILON EMPTY where
  nextState t EPSILON = EMPTY

instance (EQUAL s t b, IF b EPSILON EMPTY r)
           => NEXTSTATE t (ATOM s) r where
  nextState t (ATOM s) = cond (equal s t) EPSILON EMPTY

instance (FINALSTATE r1 b,
          NEXTSTATE t r2 rb,
          NEXTSTATE t r1 ra',
          SEQ' ra' r2 ra,
          ALT' ra rb rb'
          IF b ra rb' r)
           => NEXTSTATE t (SEQ r1 r2) r where
  nextState t (SEQ r1 r2) =
    let ra = seq' (nextState t r1) r2
        rb = nextState t r2
    in  cond (finalState r1) ra (alt' ra rb)

instance (NEXTSTATE t r1 r1',
          NEXTSTATE t r2 r2',
          ALT' r1' r2' r')
           => NEXTSTATE t (ALT r1 r2) r' where
  nextState t (ALT r1 r2) =
    alt' (nextState t r1) (nextState t r2)

instance (NEXTSTATE t r r',
```

```
          SEQ' r' (STAR r) r'')
           => NEXTSTATE t (STAR r) r'' where
  nextState t (STAR r) =
    seq' (nextState r) (STAR r)
```

Here is the definition of FINALSTATE in the traditional syntax:

```
class FINALSTATE s t | s -> t where
  finalState :: s -> t

instance FINALSTATE EMPTY FALSE
instance FINALSTATE EPSILON TRUE
instance FINALSTATE (ATOM t) FALSE
instance (FINALSTATE r r', FINALSTATE s s', AND r' s' u')
           => FINALSTATE (SEQUENCE r s) u'
instance (FINALSTATE r r', FINALSTATE s s', OR  r' s' u')
           => FINALSTATE (UNION r s) u'
instance FINALSTATE r r' => FINALSTATE (PLUS r) r'
instance FINALSTATE (STAR r) TRUE
instance FINALSTATE (OPTION r) TRUE
```

The definition of the IF class in the traditional syntax:

```
class IF b t1 t2 r | b t1 t2 -> r where
  cond :: b -> t1 -> t2 -> r

instance IF TRUE t1 t2 t1 where
  cond TRUE t1 t2 = t1
instance IF FALSE t1 t2 t1 where
  cond FALSE t1 t2 = t2
```

The definition of FINALSTATE requires type-level versions of the logical operations && and ||. These are two type classes AND and OR. Their definition in the traditional syntax:

```
class OR s t u | s t -> u
instance OR FALSE FALSE FALSE
instance OR FALSE TRUE  TRUE
instance OR TRUE  FALSE TRUE
instance OR TRUE  TRUE  TRUE

class AND s t u | s t -> u
instance AND FALSE FALSE FALSE
instance AND FALSE TRUE  FALSE
instance AND TRUE  FALSE FALSE
instance AND TRUE  TRUE  TRUE
```