Pattern Synonyms (Extended version)

Matthew Pickering
University of Oxford
Oxford, UK
matthew.pickering@cs.ox.ac.uk

Gergő Érdi
Standard Chartered Bank *
Singapore
gergo@erdi.hu

Simon Peyton Jones
Microsoft Research
Cambridge, UK
simonpj@microsoft.com

Richard A. Eisenberg
Bryn Mawr College
Bryn Mawr, PA, USA
rae@cs.brynmawr.edu

Abstract

Pattern matching has proven to be a convenient, expressive way of inspecting data. Yet this language feature, in its traditional form, is limited: patterns must be data constructors of concrete data types. No computation or abstraction is allowed. The data type in question must be concrete, with no ability to enforce any invariants. Any change in this data type requires all clients to update their code.

This paper introduces *pattern synonyms*, which allow programmers to abstract over patterns, painting over all the shortcomings listed above. Pattern synonyms are assigned types, enabling a compiler to check the validity of a synonym independent of its definition. These types are intricate; detailing how to assign a type to a pattern synonym is a key contribution of this work. We have implemented pattern synonyms in the Glasgow Haskell Compiler, where they have enjoyed immediate popularity, but we believe this feature could easily be exported to other languages that support pattern matching.

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

Keywords Haskell, pattern matching, functional programming

1. Introduction

You are writing a prototype type-checker for your new compiler, so you need a simple structure for your types. Here is one possibility:

$$\textbf{type}\ \mathit{TyConName} = \mathit{String}$$
$$\textbf{data}\ \mathit{Type} = \mathit{TyApp}\ \mathit{TyConName}\ [\mathit{Type}]$$

The type $\mathit{Int} \to \mathit{Int}$ would thus be represented like this:

$$\mathit{TyApp}\ \texttt{"->"}\ [\mathit{TyApp}\ \texttt{"Int"}\ [], \mathit{TyApp}\ \texttt{"Int"}\ []]$$

Building values like this is tiresome, so we can use a function:

$$\mathit{mkFunTy}\ t1\ t2 = \mathit{TyApp}\ \texttt{"->"}\ [t1, t2]$$
$$\mathit{intTy} = \mathit{TyApp}\ \texttt{"Int"}\ []$$

Now we can write $(\mathit{intTy}\ `\mathit{mkFunTy}`\ \mathit{intTy})$ to conveniently construct the type. But what about pattern matching? If we want to decompose a Type, Haskell forces you do to so concretely, like this

$$\mathit{funArgTy} :: \mathit{Type} \to \mathit{Type}$$
$$\mathit{funArgTy}\ (\mathit{TyApp}\ \texttt{"->"}\ [t1, \_]) = t1$$

We would prefer to abstract the details away and write

$$\textbf{pattern}\ \mathit{FunTy}\ t1\ t2 = \mathit{TyApp}\ \texttt{"->"}\ [t1, t2]$$
$$\mathit{funArgTy}\ (\mathit{FunTy}\ t1\ \_) = t1$$

defining FunTy as a synonym for the TyApp application, and using it as a pattern in $\mathit{funArgTy}$.

This paper describes one way to add support for pattern abstraction in Haskell. Allowing this new form of abstraction enables programmers to capitalise on one of the great successes of modern, typed functional programming languages: pattern matching. Defining a function via pattern matching recalls programming's mathematical roots, is declarative, and is universal within the Haskell ecosystem; it is thus prudent to build upon this success. Abstracting over patterns is a well-studied problem (§9), but our implementation experience threw up a long series of unexpected challenges. Our contribution is not simply the idea of abstracting over patterns, but rather the specifics of its realisation in a full-scale language and optimising implementation. In particular, we are the first to deal, in full generality, with pattern abstraction over patterns that involve existentials, GADTs, provided constraints, and required constraints. Our contributions are these:

- We describe *pattern synonyms*, a complete, fully implemented, and field-tested extension to the Haskell language.
- Our design accommodates both unidirectional and bidirectional patterns (§4.1-4.3), named fields (§4.4), infix synonyms (§4.5), and a "bundling" mechanism for import/export (§4.7).
- Uniquely, our design also accommodates Haskell and GHC's numerous extensions to basic pattern matching. The features that have a major impact are: view patterns and overloaded literals (§3.1), existentials (§3.2), and GADTs (§3.3).
- We provide a precise semantics for pattern synonyms, subtly different than those defined by macro expansion (§5).
- We designed and implemented *pattern signatures* that play the same role for pattern synonyms that ordinary type signatures do for functions (§6). A pattern signature needs more structure than a value signature, something we did not at first realise.
- We describe a modular implementation that supports separate compilation, and yet, for simple patterns, compiles just as efficiently as direct concrete pattern matching (§7).
- We discuss the usage that our implementation[1] has seen in real-world Haskell libraries (§8).

There is a rich literature of related work, as we discuss in §9.

* Gergő Érdi is employed by Standard Chartered Bank. This paper has been created in a personal capacity and Standard Chartered Bank does not accept liability for its content. Views expressed in this paper do not necessarily represent the views of Standard Chartered Bank.

[1] The details in this paper pertain to GHC 8.0.1 and above. Releases since GHC 7.8 support pattern synonyms, but several details have evolved since.

2. Motivation

We first describe three use cases for pattern synonyms.

2.1 Abstraction

As described in the introduction the primary motivation for pattern synonyms is *abstraction*. We have seen a simple example there. Here is a slightly more involved one.

Consider the abstract data type *Seq* which represents double-ended sequences[2]. It provides efficient support for many more operations than built-in lists but at the cost of a more complicated internal representation. It is common to see code which uses view patterns and the provided view functions in order to simulate pattern matching. With pattern synonyms, we can do much better.

The library provides a view function *viewl* which projects a sequence to a data type which allows a user to inspect the left-most element. Using this function we can define a collection of pattern synonyms *Nil* and *Cons* which can be used to treat *Seq* as if it were a cons list.

```
data ViewL a = EmptyL | a :< Seq a
viewl :: Seq a → ViewL a

pattern Nil :: Seq a
pattern Nil ← (viewl → EmptyL) where
  Nil = empty

pattern Cons :: a → Seq a → Seq a
pattern Cons x xs ← (viewl → x :< xs) where
  Cons x xs = x <| xs
```

In the code above, we use *explicitly bidirectional* pattern synonyms (§4.3), which can have a different meaning as a pattern (as specified after the ← on the first line of the definitions) and as an expression (as specified after the **where**). Being able to specify the pattern meaning and the expression meaning separately is critical here as a view pattern cannot ever be used in an expression.

We don't dwell further here as this desire is extensively studied. Many more motivating examples can be found in prior work including that of Wadler (1987) and Burton et al. (1996).

2.2 Readability

Pattern synonyms can also be used to improve readability. DSH[3] (Database-Supported Haskell) is a real-world example of a package which uses pattern synonyms to do so. DSH provides a deeply-embedded domain-specific language and also an optimising query compiler. There is a simple core representation for expressions. One of the compiler passes is a normalisation step which rewrites expressions into a normal form. As the core data type is quite simple, it would be opaque to write the normalisation rules in terms of the basic constructors. The authors instead choose to define pattern synonyms which reduce the verbosity by giving names to complex patterns.

One of their rewrite rules is to replace $length\ xs \equiv 0$ with $null\ xs$. We can define special patterns which correspond to equality expressions, application of the *length* function, and the constant 0. The following example is slightly modified from their code:

```
pattern e1 :== e2 ← BinOp _ (SBRelOp Eq) e1 e2
pattern PLength e ← AppE1 _ Length e
pattern Zero ← Lit _ (ScalarV (IntV 0))
```

With these pattern synonyms, the normalisation step becomes much easier to read.

```
normalise e =
  case e of
    PLength xs :== Zero → mkNull xs
    ...
```

Using pattern synonyms to give names to complex patterns is an important analogue to naming complex expressions. Giving names signals intent and allows users to write self-documenting code.

2.3 Polymorphic Pattern Synonyms

By using type classes and view patterns, it is possible to define pattern synonyms which work for many different data types. We call these *polymorphic* pattern synonyms due to the fact that the head of the type of the scrutinee is a type variable. Usually pattern synonyms are monomorphic in the sense that the head is fixed at definition to a particular type constructor.

We can define a type class which includes only the necessary methods to implement a particular abstract specification for a data type. Programs can be written against this interface and the concrete representation chosen later. The main advantage of this approach is the *appearance* of a normal Haskell program. This style of programming has long been possible but pattern synonyms add a convenient gloss to hide necessary internal plumbing. The following example is modified from Burton et al. (1996).

```
class ListLike f where
  nil   :: f a
  cons :: a → f a → f a
  nilMatch   :: f a → Bool
  consMatch :: f a → Maybe (a, f a)

pattern Nil :: ListLike f ⇒ f a
pattern Nil ← (nilMatch → True) where
  Nil = nil

pattern Cons :: ListLike f ⇒ a → f a → f a
pattern Cons x xs ← (consMatch → Just (x, xs))
  where
    Cons x xs = cons x xs
```

The methods in *ListLike* provide exactly the parts necessary to implement *Nil* and *Cons*. We can then implement this class for many different definitions of lists.

```
type DList a = [a] → [a]
data SnocList a = Empty | Snoc (SnocList a) a
data JoinList a
   = JNil | Unit a | JoinList a `JoinTree` JoinList a

instance ListLike [] where ...
instance ListLike DList where ...
instance ListLike JoinList where ...
```

Using *Cons* and *Nil* we can write programs which are generic in the type of list representation we choose.

```
listLength :: ListLike f ⇒ f a → Int
listLength Nil = 0
listLength (Cons _ xs) = 1 + listLength xs
```

3. Pattern Matching in Haskell

Before we can introduce our extension, we first describe our baseline: pattern matching in Haskell. Haskell pattern matching, and GHC's extensions thereof, lead to a number of challenges for our goal of abstraction.

Figure 1 gives the syntax for the subset of Haskell patterns that we treat in this paper. Variables and constructor patterns are conventional; we will discuss view patterns in §3.1. The actual

[2] We specifically consider the implementation provided in the containers package – `https://hackage.haskell.org/package/containers-0.5.7.1/docs/Data-Sequence.html`

[3] `https://hackage.haskell.org/package/DSH-0.12.0.1`

var, x	$::= \ldots$	Variable name
$conid, K$	$::= \ldots$	Data constructor name
$conop$	$::= \ldots$	Data constructor operator
$expr$	$::= \ldots$	Expression
pat	$::= var$	Variable pattern
	$\mid$ '_'	Wildcard pattern
	$\mid$ $conid\ pat_1 \ldots pat_n$	Constructor pattern
	$\mid$ $pat_1\ conop\ pat_2$	Infix constructor pattern
	$\mid$ $expr$ '$\rightarrow$' pat	View pattern

Figure 1. The syntax of patterns in Haskell

syntax is much richer, including literals, as-patterns, tuples, etc., but it is all syntactic sugar and this subset exposes all the challenges.

3.1 View Patterns and Literal Patterns

View patterns (Figure 1) are an extension to the basic pattern matching syntax which allows computation to be performed during pattern matching. A view pattern $f \rightarrow p$ consists of two parts – a function f (commonly referred to as the view function) and a pattern p. When a value v is matched against this view pattern, we compute $(f\ v)$, and match the result against the pattern p. For example:

$$
\begin{aligned}
&uncons :: [a] \rightarrow Maybe\ (a, [a]) \\
&safeHead :: [a] \rightarrow Maybe\ a \\
&safeHead\ (uncons \rightarrow Just\ (x, \_)) = Just\ x \\
&safeHead\ \_ = Nothing
\end{aligned}
$$

In this example the view function is $uncons$, a function which tries to detach the first element from a list. By using it inside a view pattern, we can then match on the $Just$ constructor of the result of the computation.

Although view patterns are a GHC extension, they are simply a generalisation of standard Haskell's overloaded numeric literal patterns. For example:

$$
\begin{aligned}
&f :: (Eq\ a, Num\ a) \Rightarrow a \rightarrow a \\
&f\ 0 = 0 \\
&f\ n = n + 1
\end{aligned}
$$

means the same as

$$
\begin{aligned}
&f\ ((\equiv 0) \rightarrow True) = 0 \\
&f\ n = n + 1
\end{aligned}
$$

The view pattern compares the value with zero (using whatever definition for $\equiv$ has been supplied with the argument's Eq instance), and matches the result of that comparison with $True$.

Notice that f works only on argument types that are in class Num (so that we can build a zero value) and Eq. In general, a view pattern may require an arbitrary set of type classes. For example, consider this:

$$
\begin{aligned}
&foo :: (Ord\ t, Bounded\ t) \Rightarrow Tree\ t \rightarrow Maybe\ t \\
&g :: (Ord\ b, Bounded\ b) \Rightarrow ([a], Tree\ b) \rightarrow (a, b) \\
&g\ ([x], foo \rightarrow Just\ y) = (x, y)
\end{aligned}
$$

Since the view function foo requires $(Ord\ t, Bounded\ t)$, so does g. These are the *required constraints*, a concept that will become more important in the context of pattern synonyms; see §6.1.

3.2 Pattern Matching with Existentials

In Haskell a data constructor may bind an existential type variable. For example:

$$
\begin{aligned}
&\textbf{data}\ T\ \textbf{where} \\
&\quad MkT :: a \rightarrow (a \rightarrow Int) \rightarrow T \\
&f :: T \rightarrow Int \\
&f\ (MkT\ x\ g) = g\ x + 1 \\
&foo :: [Int] \\
&foo = map\ f\ [MkT\ \texttt{'c'}\ ord, MkT\ 5\ id]
\end{aligned}
$$

The data constructor MkT binds, essentially, three pieces of information: the type chosen for a, a value of type a, and a function that can convert a value of type a into an Int. Accordingly, given two values of type T, the types of the data inside might be different, as the caller of the MkT constructor can choose any type for a. Pattern matching on MkT, such as in function f, brings $(x :: a)$ and $(g :: a \rightarrow Int)$ into scope, where the type variable a is (invisibly) bound by the same pattern.

The data constructor may also bind type class constraints. For example:

$$
\begin{aligned}
&\textbf{data}\ T2\ \textbf{where} \\
&\quad MkT2 :: Show\ a \Rightarrow a \rightarrow (a \rightarrow Int) \rightarrow T2 \\
&f2 :: T2 \rightarrow String \\
&f2\ (MkT2\ x\ g) = show\ x +\!\!+ show\ (g\ x) \\
&foo2 :: [String] \\
&foo2 = map\ f2\ [MkT2\ \texttt{'c'}\ ord, MkT2\ 5\ id]
\end{aligned}
$$

Now $MkT2$ wraps up four data: the type a, a value of type a, a function of type $a \rightarrow Int$, *and* the $Show\ a$ dictionary[4]. When $f2$ matches against $MkT2$, the $Show$ operations become available inside the match; hence our ability to call $(show\ x)$ in $f2$'s right hand side. Notice that there is no $Show$ constraint in $f2$'s type; rather it is provided by the pattern match.

3.3 GADT Pattern Matching

With GHC's $GADTs$ extension, pattern matching can also perform local type refinement (Peyton Jones et al. 2006). For example:

$$
\begin{aligned}
&\textbf{data}\ Expr\ a\ \textbf{where} \\
&\quad IntExpr :: Int \rightarrow Expr\ Int \\
&\quad \ldots \\
&addToExpr :: Expr\ a \rightarrow a \rightarrow a \\
&addToExpr\ (IntExpr\ i)\ j = i + j
\end{aligned}
$$

By matching on $IntExpr$ we refine the type variable a to Int (as $IntExpr\ i :: Expr\ Int$). Thus, the second argument is now also known to be of type Int. We can thus add the two arguments in order to produce a result of type Int.

However, this is just convenient syntactic sugar for the ability of data constructors to bind type constraints. The data type $Expr$ can equivalently be written[5]

$$
\begin{aligned}
&\textbf{data}\ Expr\ a\ \textbf{where} \\
&\quad IntExpr :: (a \sim Int) \Rightarrow Int \rightarrow Expr\ a \\
&\quad \ldots
\end{aligned}
$$

So, just as described in the previous section, pattern matching against $IntExpr$ will provide the constraint $(a \sim Int)$ to the body of the pattern match.

4. Pattern Synonyms

In this section we introduce our new extension, *pattern synonyms*. The new syntax is given in Figure 2. We add one new top-level declaration form, introduced by the keyword **pattern**. We make no changes to the syntax of patterns.

[4] A *dictionary* is GHC's runtime witness for class constraints. It is essentially a record containing implementations of all of the methods in the $Show$ class.

[5] Here, we are using Haskell's $(\sim)$ type equality operator.

psyn	:= ‘**pattern**’ *lhs* ‘=’ *pat*	Implicitly bidir.
	| ‘**pattern**’ *lhs* ‘←’ *pat*	Unidirectional
	| ‘**pattern**’ *lhs* ‘←’ *pat* ‘**where**’ *lhs* ‘=’ *expr*	Explicitly bidir.
lhs	::= *conid* $var_1 \ldots var_n$	Prefix notation
	| var_1 *conop* var_2	Infix notation
	| *conid* ‘{’ var_1 ‘,’ … ‘,’ var_n ‘}’	Record notation
sig	::= ‘**pattern**’ *conid* ‘::’ *pat_ty*	Pattern signature
pat_ty	::= *req prov type*	Pattern type
req	::= ⟨*empty*⟩ | *ctx* ‘⇒’	Required context
prov	::= ⟨*empty*⟩ | *ctx* ‘⇒’	Provided context

Figure 2. The syntax of pattern synonym declarations.

4.1 Implicitly Bidirectional Patterns

The simplest form of pattern synonym is an *implicitly bidirectional* pattern synonym, written with a "=" sign:

```
pattern Just2 a b = Just (a, b)

f :: Maybe (Int, Int) → Int
f (Just2 x y) = x + y
foo :: Int
foo = f (Just2 3 2)
```

The **pattern** declaration introduces the pattern synonym *Just2*. It can be used as a pattern (e.g. function f); matching on *Just2* x y will behave like matching on the nested pattern *Just* (x, y). Like a data constructor, *Just2* can also be used in an expression (e.g. in *foo*); the expression *Just2* e_1 e_2 behave like the expression *Just* (e_1, e_2).

Several points are worth noting:

- Just like data constructors, pattern synonyms must start with a capital letter or ":". This is why we continue to use the non-terminal *conid* in Figure 2.
- The pattern on the right hand side must bind exactly the same variables as those bound on the left hand side.
- The right hand side can, of course, be nested to arbitrary depth, and can use pattern synonyms. For example:

```
pattern Complex :: a → b → c
                → (Maybe ([a], b), String, c)
pattern Complex a b c = (Just2 [a] b, "2", c)
```

- Type inference works for pattern synonym declarations just as for value declarations, so GHC infers

  ```
  pattern Just2 :: a → b → Maybe (a, b)
  ```

  Programmers can also specify an explicit type signature for the pattern synonym. Such signatures are called *pattern signatures*, and we discuss them in more detail in §6. Meanwhile we will often use pattern signatures informally.

4.2 Unidirectional Pattern Synonyms

In a bidirectional pattern synonym such as *Just2*, the right-hand side, *Just* (a, b), is used both as a pattern (when matching) and as an expression (when constructing). This works nicely because many Haskell patterns, by design, look like expressions. But not all! What would this declaration mean?

```
pattern Head x = x : _   -- Not right yet
```

Here, *Head* would make sense as a pattern, but not in an expression. What would it mean to say

```
foo = Head 3   -- Bizarre??
```

View patterns are an even clearer example, where it makes no sense to interpret the pattern as an expression.

Nevertheless it is often convenient to define *unidirectional* pattern synonyms like *Head*, which we indicate using "←" instead of "=" in the declaration

```
pattern Head :: a → [a]
pattern Head x ← x : _   -- Unidirectional
```

The back-arrow arrow indicates that *Head* can be used in patterns, but not in expressions. Because of this, the lexical scoping rule for unidirectional pattern synonyms is relaxed: the pattern on the right must bind all the variables bound on the left, but it is free to bind more.

4.3 Explicitly Bidirectional Pattern Synonyms

Sometimes, however, you really want a bidirectional pattern synonym (i.e. one that you can use in both patterns and expressions) where there is computation to do in both directions. A classic example is the conversion from rectangular to polar coordinates:

```
data Point = CP Float Float   -- x,y coords
pointPolar :: Point → (Float, Float)
pointPolar (CP x y) = ...
polarPoint :: Float → Float → Point
polarPoint r a = ...
pattern Polar :: Float → Float → Point
pattern Polar r a ← (pointPolar → (r, a))
  where
    Polar r a = polarPoint r a
```

Here, the data type *Point* is represented using Cartesian coordinates. *pointPolar* and *polarPoint* are (ordinary Haskell) functions which allow points to be viewed and constructed, respectively, using polar coordinates. Finally, the pattern synonym *Polar* uses a view pattern to match a *Point*, using *pointPolar* to extract its polar coordinates.

The new piece is the **where** clause on the **pattern** declaration, which signals an *explicitly bidirectional* pattern synonym. It allows the programmer to specify how *Polar* should behave when used in an expression. The **where** clause has a single definition that looks exactly like a normal function definition, except that the function being defined, *Polar*, starts with a capital letter. Note that the variables bound in the first line (r and a) are not in scope in the **where** clause; there is no name shadowing in this example. A **where** clause is not allowed for an implicitly bidirectional synonym (§4.1).

We refer to the definition in the **where** clause as the *builder*, while the pattern after the "←" is the *matcher*. Apart from having to have the same type, there is no required relationship between the matcher and the builder. One might expect that they should be inverses of each other, but we do not impose such a restriction. Indeed, real world usage has demonstrated how this asymmetry can be used to great effect. For one example, see (Jaskelioff and Rivas 2015).

Of course, if this transformation is expensive then pattern matching becomes very expensive. Programmers are used to pattern matching being a very cheap operation and might be surprised when an innocuous-looking program has poor performance. This potential trap is inevitable with the design, where we are explicitly providing succinct syntax for arbitrary operations.

4.4 Pattern Synonyms with Named Fields

Standard Haskell data type declarations allow the fields of a data constructor to be named, thus:

```
data Point = CP {x :: Float, y :: Float}
diagX :: Point → Point
diagX (CP {x = xp}) = CP {x = xp, y = xp}
```

You can use these field names in both pattern matching and construction, as shown in function $diagX$. The advantage is that it makes the program more robust to changes in the data type declaration, such as adding fields, or changing their order. Note that CP still has its ordinary curried type $CP :: Float \to Float \to Point$, and can also be called with positional arguments, e.g. $(CP\ xp\ yp)$.

We extended this named-field facility to pattern synonyms. Referring to the syntax in Figure 2, we can define the $Polar$ pattern synonym of §4.3 like this:

```
pattern Polar :: Float → Float → Point
pattern Polar {r, a} ← (pointPolar → (r, a))
  where
    Polar rp ap = polarPoint rp ap
```

The curly braces on the left hand side of the pattern synonym declaration enclose a list of the named fields. Like ordinary data constructors, $Polar$ can still be used with positional arguments, with the order being specified by the list of field names in the braces. This named-field facility can be used for a pattern synonym of any directionality.

In GHC, data constructors with named fields can be used in no fewer than nine different ways, listed in Figure 3. Only the first two apply to data constructors without named fields. The extra seven forms are supported uniformly for pattern synonyms with named fields, because they all have simple desugarings into the basic first two. For example these four definitions all mean the same thing, with (5,7,9) all desugaring to (2).

```
getR1 (Polar r _)   = r        -- (2)
getR2 (Polar {r = r}) = r      -- (5)
getR3 (Polar {r})   = r        -- (7)
getR4 (Polar {..}) = r         -- (9)
```

Even record update extends uniformly (for bidirectional pattern synonyms), because the record-update expression $(r\ \{x = e\})$ desugars to

```
case r of
  Polar {x = x, y = y} → Polar {x = e, y = y}
```

4.5 Infix Pattern Synonyms

Lexically, pattern synonyms behave just like data constructors, and live in the same name-space. In particular,

- A pattern synonym can be an infix operator symbol starting with ":"; for example:

```
infixl 4 :<|    -- Optional fixity declaration
pattern x :<| xs ← (viewl → x :< xs) where
  x :<| xs = x <| xs
```

- A non-operator pattern synonym can be turned into an infix operator, in both patterns and expressions, by surrounding it with backticks, thus:

```
f (x `Just2` y) = y `Just2` x
```

4.6 Pattern Synonyms with a Non-Data Return Type

A pattern synonym usually matches values of some particular *data* type. For example:

```
pattern P :: a → Maybe (Maybe a)
pattern P a = Just (Just a)
```

which matches values of type $Maybe\ (Maybe\ t)$. But pattern synonyms can also be polymorphic in their return type, as we saw in §2.3:

```
pattern Cons :: ListLike f ⇒ a → f a → f a
```

Pattern synonyms can even match a *function* value. For example:

```
pattern F :: a → (Bool → a)
pattern F a ← (($ True) → a)
```

The rather strange view pattern matches against a function; it applies that function to $True$ and binds the result to a. So these two functions are equivalent:

```
f1, f2 :: (Bool → [Int]) → Int
f1 (F xs) = sum xs
f2 g = let xs = g True in sum xs
```

4.7 Import and Export of Pattern Synonyms

We also need to consider how to import and export pattern synonyms. Before our implementation, names in the data constructor namespace (and thus pattern synonyms) could be exported only with the parent type constructor. As pattern synonyms have no parent type constructor, there was no existing mechanism to export pattern synonyms.

4.7.1 Independently Exporting Names

Users may export and import pattern synonyms individually. A new keyword **pattern** is used to allow identifiers in the constructor namespace to appear in import and export lists. If **pattern** *Foo* appears in an export list, then the pattern synonym *Foo* will be exported from the module. (Without the **pattern** modifier, *Foo* would look like a type.)

```
module M (pattern Foo) where ...
pattern Foo ...
```

Likewise, when importing M, users may explicitly import *Foo* with an import statement of the same form.

```
module N where
import M (pattern Foo)
```

4.7.2 Bundling

Pattern synonyms permit a library author the freedom to change out a concrete representation of a datatype, even when the datatype's constructors were exported. Simply replace all constructors with pattern synonyms that resolve to use the new internal representation, and the downstream code continues to work. The only hiccup in this plan is around import/export lists, where a user expects $T\ (..)$ to affect the datatype T with all of its constructors. Our solution is called *bundling*.

A user can bundle a pattern synonym with a type constructor in the export list of a module. Once bundled, a pattern synonym behaves like a child data constructor. In order to bundle a pattern synonym P with T, we just need to list it explicitly in the export list of the type constructor T.

```
module M (T (P)) where
data T = MkT
```

	Context Type	Description	GHC extension	Example
(1)	E	As an expression		$zero = CP\ 0\ 0$
(2)	P	In a pattern		$isZero\ (CP\ 0\ 0) = True$
(3)	E	Using record update syntax		$p\ \{x = 1\}$
(4)	E	As an expression with record syntax		$zero = CP\ \{x = 0, y = 0\}$
(5)	P	In a pattern with record syntax		$isZero\ (CP\ \{x = 0, y = 0\}) = True$
(6)	E	In an expression with field puns	*NamedFieldPuns*	$\mathbf{let}\ x = 5; y = 6\ \mathbf{in}\ CP\ \{y, x\}$
(7)	P	In a pattern with field puns	*NamedFieldPuns*	$getX\ (CP\ \{x\}) = x$
(8)	E	In an expression with record wildcards	*RecordWildCards*	$\mathbf{let}\ x = 5; y = 6\ \mathbf{in}\ CP\ \{..\}$
(9)	P	In a pattern with record wildcards	*RecordWildCards*	$getX\ (CP\ \{..\}) = x$

Figure 3. The different ways a record constructor can be used

$\mathbf{pattern}\ P :: T$
$\mathbf{pattern}\ P = MkT$

In modules importing M, one can use wildcard notation in order to import all the children of T, of which P is now one.

$\mathbf{module}\ N\ \mathbf{where}$
$\mathbf{import}\ M\ (T\ (..))$

Imports of this form will also import P as it is bundled with T on its export in M.

A potential drawback of this feature is that users may unwittingly use pattern synonyms where they are expecting ordinary data constructors. These pattern synonyms might unexpectedly be expensive at runtime. However, it is up to the library author not to surprise her clients by an abuse of bundling. Pattern synonyms remain documented differently from ordinary data constructors in Haddock documentation, regardless of bundling.

When pattern synonyms are exported in this way, we perform some basic typechecking in order to catch obvious mistakes. If the scrutinee type of a pattern synonym P has a definite type constructor T then it can be bundled only with T. However, a pattern synonym which is polymorphic in the constructor (such as in §2.3) can be bundled with any type constructor as there is no clear specification for when we should reject such associations.

$\mathbf{class}\ C\ f\ \mathbf{where}$
  $get :: f\ a \to a$
$\mathbf{pattern}\ Poly :: C\ f \Rightarrow a \to f\ a$
$\mathbf{pattern}\ Poly\ v \leftarrow (match \to v)$

When should we allow $Poly$ to be associated with a type constructor T? Should we require that T is an instance of C? No, because an orphan instance could be available at the use site but not at the definition site. However, it might be the case that it is impossible to define such an instance for T (say T has the wrong kind). In these situations it would be desirable to disallow such exports. Further, the contexts may include equality constraints and class constraints may have superclass constraints (which may also include equality constraints!). Hence, *prima facie*, we can not decide easily whether bundling should be permitted. As this decision doesn't affect soundness we allow any such association for this more complex sort of pattern synonym.

5. Semantics of Pattern Synonyms

It is important to be able to give a precise, modular semantics to pattern synonyms. Our baseline is the semantics of pattern matching given in the Haskell 2010 report (Marlow et al. 2010), which is defined as follows.

When matching a pattern against a value, there are three possible outcomes: the match *succeeds* (binding some variables); the match *fails*; or the match *diverges*. Failure is not an error: if the match fails in a function defined by multiple equations, or a case expression with multiple alternatives, we just try the next equation.

The semantics of matching a pattern p against a value v is given by cases on the form of the pattern:

1. If the pattern is a variable x then the match always succeeds, binding x to the value v.
2. If the pattern is a wildcard '_' then the match always succeeds, binding nothing.
3. If the pattern is a constructor pattern $(K\ p_1\ ...\ p_n)$, where K is a data constructor, then:
   (a) If the value v diverges (is bottom), then the match diverges.
   (b) If the value is of the form $(K\ v_1 ... v_n)$ then we match the sub-patterns from left-to-right (p_1 against v_1 and so on). If any of the patterns fail (or diverge) then the whole pattern match fails (or diverges). The match succeeds if all sub-patterns match.
   (c) If the value is of the form $(K'\ v_1\ ...\ v_n)$, where $K \neq K'$ then the match fails.

It is simple to extend this semantics to cover view patterns and pattern synonyms, as follows

4. If the pattern is a view pattern $(f \to p)$ then evaluate $(f\ v)$. If evaluation diverges, the match diverges. Otherwise, we match the resulting value against p.
5. If the pattern is a constructor pattern $(P\ p_1\ ...\ p_n)$, where P is a pattern synonym defined by $P\ x_1\ ...\ x_n = p$ or $P\ x_1\ ...\ x_n \leftarrow p$, then:
   (a) Match the value v against p. The pattern p must bind the variables x_i; let them be bound to values v_i. If this match fails or diverges, so does the whole (pattern synonym) match.
   (b) Match v_1 against p_1, v_2 against p_2 and so on. If any of these matches fail or diverge, so does the whole match.
   (c) If all the matches against the p_i succeed, the match succeeds, binding the variables bound by the p_i. (The x_i are not bound; they remain local to the pattern synonym declaration.)

These definitions are simple and modular, but it is worth noting that the semantics of matching a pattern synonym is subtly different from macro-substitution. Consider:

$f1 :: (Maybe\ Bool, Maybe\ Bool) \to Bool$
$f1\ (Just\ True, Just\ False) = True$
$f1\ \_ = False$
$\mathbf{pattern}\ Two\ ::\ Bool \to Bool$
$\qquad\qquad \to (Maybe\ Bool, Maybe\ Bool)$
$\mathbf{pattern}\ Two\ x\ y = (Just\ x, Just\ y)$

$f2 :: (Maybe\ Bool, Maybe\ Bool) \to Bool$
$f2\ (Two\ True\ False) = True$
$f2\ \_ = False$

On non-bottom values, *f1* and *f2* behave identically. But the call $(f1\ \ (Just\ \bot, Nothing))$ will diverge when matching $(Just\ \bot)$ against the pattern $(Just\ True)$. On the other hand if *f2* is applied to the same value, using the semantics above, matching the value against the pattern $(Two\ True\ False)$ starts by matching against the right hand side of Two's definition, $(Just\ x, Just\ y)$, which fails! So the second equation for *f2* is tried, which succeeds, returning $False$.

Although a little unexpected, we are convinced that this is the right choice, because it allows the semantics to be modular. The "macro-substitution" semantics could only be explained by saying "expand out the pattern synonyms, and take the semantics of that", which would be problematic for pattern synonyms that use further pattern synonyms, and so on. Moreover, the implementation would also be forced to use macro-expansion, with resulting code duplication.

6. Pattern Signatures

Just as Haskell gives a type to every function and data constructor, we must have a type for every pattern synonym P. The whole point of having a type system is embodied in the following principle:

> *The Typing Principle*. It should be possible to determine whether a use of P is well-typed based only on P's type, without reference to P's declaration.

This principle – which often goes unstated – is helpful to have written explicitly, as it is much harder to achieve than one might suppose.

6.1 Required and Provided Constraints

Recall from Section 3 the following:

- Type constraints may be *required* by a view pattern. These constraints must be satisfied in order for the pattern match to be well-typed.
- Type constraints may be *provided* by a constructor pattern. These constraints are bound by a successful pattern match and are considered satisfiable in the context of the match.

A complex pattern, containing both constructors and literal/view patterns, may *both* require some constraints, *and* provide others. For example:

$\mathbf{data}\ RP\ a\ \mathbf{where}$
$\quad MkRP :: (Eq\ a, Show\ b) \Rightarrow a \to b \to RP\ a$
$f\ (MkRP\ 3\ v) = show\ v$

The pattern $(MkRP\ 3\ v)$ repays careful study. Matching $MkRP$ provides $(Eq\ a, Show\ b)$. Then matching the nested literal pattern 3 requires $(Eq\ a, Num\ a)$, and the use of $show$ on the RHS requires $(Show\ b)$. Of these, $(Eq\ a)$ and $(Show\ b)$ are both satisfied by the constraints provided by the match on $MkRP$, which leaves $(Num\ a)$ as required. So, f's inferred type is

$f :: Num\ a \Rightarrow RP\ a \to String$

To summarise, the pattern $(MkRP\ 3\ v)$ *provides* $(Eq\ a, Show\ b)$, and *requires* $(Num\ a)$.

6.2 Pattern Signatures

Suppose that we want to abstract this to a pattern synonym:

$\mathbf{pattern}\ P\ v = MkRP\ 3\ v$

What type should we assign to P?[6] The key observation is this: *to uphold the Typing Principle, P's type must somehow distinguish between its provided and required constraints*. Moreover, we need a concrete syntax for P's type, so that we can write pattern signatures, and so that GHCi can display P's type.

As remarked above, P *requires* $(Num\ a)$ and *provides* the constraint $(Eq\ a, Show\ b)$. We use the following syntax to describe P's type:

$\mathbf{pattern}\ P :: Num\ a \Rightarrow (Eq\ a, Show\ b) \Rightarrow b \to RP\ a$

The general form of a pattern signature is thus (Figure 2)

$\mathbf{pattern}\ P :: required \Rightarrow provided \Rightarrow$
$\qquad arg_1 \to ... \to arg_n \to res$

That is, the required context comes first, then the provided context. We stress: this is a new element in the Haskell grammar, particular to pattern synonyms. Its syntax deliberately makes it look a bit like an ordinary Haskell type, but it is best to think of it as a new syntactic construct – a *pattern type*, if you will.

If there are no required or provided contexts, you can omit them altogether; you may also omit only the provided context. However, to avoid ambiguity, if there is a provided context but no required one, you must write () for the required one, thus:

$\mathbf{pattern}\ P :: arg \to res$ -- No contexts
$\mathbf{pattern}\ P :: req \Rightarrow arg \to res$ -- Required only
$\mathbf{pattern}\ P :: () \Rightarrow prov \Rightarrow arg \to res$ -- Provided only

In practice, the occurrence of required constraints seems more common than provided ones, so the slightly regrettable final form is relatively rare.

6.3 Universal and Existential Quantification

We introduced the concept of existential variables in §3.2; an existential type variable is brought into scope by a pattern match. Other type variables involved in a pattern match are called *universal* type variables – these are the type variables in scope outside of the match. Thus, when we match the pattern $Just\ x$ of type $Maybe\ a$, that a is a universal variable. If a variable a is universal in a pattern p, then the pattern is well-typed for *all* choices of a. If a variable b is existential in the pattern, then the pattern brings into scope *some* choice for b.

A close reading of the definitions of universals and existentials will show a striking similarity with required and provided constraints. Indeed, this similarity is telling, with universal variables being paired with required constraints (both must be available before the match) and existential variables being paired with provided constraints (both are made available by the match).

The syntax of pattern type signatures allows the programmer to explicitly mention type variables, if they so choose, using the `forall` keyword (typeset as ∀). According to the pairings above, universals appear with required constraints and existentials appear with provided ones. The example P from the beginning of the previous subsection can be given this signature:

$\mathbf{pattern}\ P\ ::\ \forall a.Num\ a$
$\qquad \Rightarrow \forall b.(Eq\ a, Show\ b)$
$\qquad \Rightarrow b$
$\qquad \to RP\ a$

We can see that a is universal and b is existential.

However, just as with constraints, figuring out which variables belong in which category is sometimes quite subtle. At first blush, it may seem that all universal variables must be mentioned in the

[6] Please see the appendix for a formalisation of pattern synonym types, which details what programs are accepted.

result type; after all, this is the case for data constructors. Yet it isn't so for pattern synonyms! Witness this example:[7]

$$readMaybe :: Read\ a \Rightarrow String \to Maybe\ a$$
$$\textbf{pattern}\ PRead\ a \leftarrow (readMaybe \to Just\ a)$$

Ponder a moment what the type of $PRead$ should be. The result type of $PRead$ must be the argument type of $readMaybe$, which is clearly $String$. The argument type of $PRead$ must be a type which has a $Read$ instance. This leads to the type signature

$$\textbf{pattern}\ PRead :: Read\ a \Rightarrow a \to String$$

In this signature, the variable a is properly universal: it must be known before the match, and it is not brought into scope by the match. And yet it does not appear in the result.

At second blush, it may seem that all universals must thus be mentioned either in the result or in some required constraint. Yet it isn't so – at least in degenerate cases. Witness this example:

$$magic :: Int \to a$$
$$magic = \bot$$
$$\textbf{pattern}\ Silly\ x \leftarrow (magic \to x)$$

The type for $Silly$ can be only

$$\textbf{pattern}\ Silly :: a \to Int$$

The question is this: is that a universally quantified or existentially quantified? A little thought reveals that it is a *universal*, as it is surely not brought into scope by the match. Here we have a universal, mentioned in neither the result type nor in the (absent) required constraint.

We wish to be able to infer the universal/existential distinction without user intervention, at least in the common case. We have thus implemented these Implicit Quantification rules, when the user does not write explicit ∀s:

- All type variables mentioned in the required constraints or in the result type are universals.
- All other type variables are existentials.

If the user *does* write explicit ∀s, we use the following scoping rules

- Existentials scope over only the provided constraints and the arguments, not over the result type. For example, this is rejected because the existential b appears in the result type $(T\ b)$:

  $$\textbf{pattern}\ P :: () \Rightarrow \forall b.b \to T\ b$$

- Existentials must not shadow universals.

Note that the existentials' scope is strictly smaller than the universals'. Indeed, this is what convinced us to put provided constraints *after* required ones.

The Implicit Quantification rules misbehave on the signature for $Silly$ above, where a would be considered to be existential. However, we believe that this kind of mistake is possible only in degenerate cases, where some view pattern uses a function that manufactures a value of some type out of thin air, like $magic$. To get $Silly$ to work, we must explicitly label the variable as universal, thus:

$$\textbf{pattern}\ Silly :: \forall a.a \to Int$$

Following the rules for required vs. provided constraints, if we have only one ∀ in a pattern type, it describes universal variables. Thus the ∀ makes a universal, and all is well again. We keep the Implicit Quantification Rule simple (as stated above); if you want something else, use explicit quantification.

[7] Examples in this discussion are taken from the discussion on GHC ticket #11224.

6.4 The Need for Pattern Signatures

As with any top-level construct, it is good practice to write a signature for a pattern synonym. But sometimes it is *necessary*. For example, take

$$\textbf{data}\ T\ a\ \textbf{where}$$
$$\quad MkT :: Bool \to T\ Bool$$
$$\textbf{pattern}\ P\ a \leftarrow MkT\ a$$

Should the typechecker infer $\textbf{pattern}\ P :: a \to T\ a$ or $\textbf{pattern}\ P :: Bool \to T\ a$? The user has to decide by providing a type signature along the definition of P. This situation mirrors exactly the problem of assigning a type to the function definition from Schrijvers et al. (2009):

$$f\ (MkT\ x) = x$$

Additionally, as with normal functions, a type signature can constrain a pattern synonym to a less polymorphic type, more suitable for the domain.

6.5 GADTs and Pattern Synonyms

The return type of a pattern synonym may have nested structure. Consider:

$$\textbf{data}\ T\ a = MkT\ a$$
$$\textbf{pattern}\ PT :: a \to T\ (Maybe\ a)$$
$$\textbf{pattern}\ PT\ x = MkT\ (Just\ x)$$

The return type of PT is $T\ (Maybe\ a)$, and this use of PT is ill-typed:

$$f :: T\ a \to Bool$$
$$f\ (PT\ \_) = True$$

It is ill-typed because the pattern $(PT\ x)$ cannot match values of *any* type $T\ a$, only those of type $T\ (Maybe\ a)$. You can easily see why by expanding the pattern synonym to get

$$f\ (MkT\ (Just\ \_)) = True$$

Notice that this is *different from (and simpler than) GADTs*. Consider

$$\textbf{data}\ S\ a\ b\ \textbf{where}$$
$$\quad MkS1 :: a \to S\ a\ (Maybe\ a)$$
$$\quad MkS2 :: b \to S\ a\ b$$
$$g :: c \to S\ c\ d \to d$$
$$g\ x\ (MkS1\ \{\}) = Just\ x$$
$$g\ \_\ (MkS2\ y) \quad = y$$

Here, matching on $MkS1$ refines the type d to $Maybe\ c$ – much more complicated than what happened with PT, above.

We conclude that, unlike GADTs, the fact that the return type of a pattern synonym has nested structure does not connote type refinement. But can we use pattern synonyms to abstract over GADTs? Yes, of course:

$$\textbf{pattern}\ PS\ x \leftarrow MkS1\ x$$

Now, what type should we attribute to PS? We cannot say

$$\textbf{pattern}\ PS :: a \to S\ a\ (Maybe\ a)$$

analogously to the signature of $MkS1$ because, as just discussed, a structured return type does not connote GADT behaviour. (Remember the Typing Principle!) So instead we must use the more explicit form of GADT typing, discussed in §3.3:

$$\textbf{pattern}\ PS :: () \Rightarrow (b{\sim}Maybe\ a) \Rightarrow a \to S\ a\ b$$

So in pattern signatures, all type-refinement behaviour is expressed explicitly with equality constraints, and not implicitly with structured return types.

The two may work in tandem. For example:

$$\textbf{pattern}\ PS2 :: () \Rightarrow (b \sim Maybe\ a) \Rightarrow a \to (Bool, S\ a\ b)$$
$$\textbf{pattern}\ PS2\ x \leftarrow (True, MkS1\ x)$$

6.6 Specialising a Pattern Signature

In Haskell, it is always OK to specify a type signature that is less polymorphic than the actual type of the function. For example:

$$rev\ [\,] = [\,]$$
$$rev\ (x : xs) = rev\ xs +\!\!+ [x]$$

Haskell considers all of these to be acceptable type signatures for rev:

$$rev :: [a] \to [a]$$
$$rev :: [Maybe\ b] \to [Maybe\ b]$$
$$rev :: Ord\ a \Rightarrow [a] \to [a]$$
$$rev :: [Int] \to [Int]$$

In the same way, a pattern signature is allowed to constrain the pattern's polymorphism. For example, any of these signatures is acceptable for the pattern synonym $Just2$:

$$\textbf{pattern}\ Just2\ x\ y = Just\ (x, y)$$
$$\textbf{pattern}\ Just2 :: a \to b \to Maybe\ (a, b)$$
$$\textbf{pattern}\ Just2 :: a \to a \to Maybe\ (a, a)$$
$$\textbf{pattern}\ Just2 :: Int \to Bool \to Maybe\ (Int, Bool)$$

This specialisation works fine for the universally-quantified type variables and required constraints of a pattern signature. But for the existential part, matters are necessarily more complicated. For example:

$$\textbf{data}\ T\ \textbf{where}$$
$$\quad MkT :: Show\ a \Rightarrow Maybe\ [a] \to T$$
$$\textbf{pattern}\ P :: () \Rightarrow \forall b.Show\ b \Rightarrow [b] \to T$$
$$\textbf{pattern}\ P\ x = MkT\ (Just\ x)$$

The signature given is the most general one and, since the pattern is a bidirectional one, only the most-general form will do. Why? Because when using P in an expression, we must construct a T-value that has a $Show\ a$ dictionary and a $Maybe\ [a]$ payload, for some type a; and when pattern matching on MkT the client cannot assume anything more than a $(Maybe\ [a])$ value with a $(Show\ a)$ dictionary available. If we make the signature either more or less polymorphic than the one above, one or other of these requirements will not be met.

If P were a *unidirectional* pattern, thus

$$\textbf{pattern}\ P\ x \leftarrow MkT\ (Just\ x)$$

then we have more wiggle room. For example, either of these signatures would be accepted:

$$\textbf{pattern}\ P :: () \Rightarrow \forall b.[b] \to T$$
$$\textbf{pattern}\ P :: () \Rightarrow \forall c.c \to T$$

Notice that this is exactly backward from the usual story: for a unidirectional pattern synonym we can make the existential part of its signature *more* polymorphic, but not *less*.

6.7 One Type Only

It seems to go without saying that a pattern synonym should have a single type, but you might actually want it to have two. Consider this pattern synonym

$$\textbf{pattern}\ P :: (Eq\ a, Num\ a) \Rightarrow Maybe\ a$$
$$\textbf{pattern}\ P = Just\ 42$$
$$f :: (Eq\ a, Num\ a) \Rightarrow Maybe\ a \to Bool$$
$$f\ P = True$$
$$g1\ x = P$$
$$g2 :: Num\ a \Rightarrow b \to Maybe\ a$$
$$g2\ x = Just\ 42$$

Matching P requires $(Eq\ a, Num\ a)$; hence the pattern signature P, and the type signature for f that matches on P. But what about function $g1$? If we replaced P by its definition we would get $g2$, which only requires $(Num\ a)$. But function $g1$ uses pattern P, and by the Typing Principle, we must type it looking only at P's signature, not its definition. So we choose to give $g1$ the type $g1 :: (Eq\ a, Num\ a) \Rightarrow b \to Maybe\ a$.

An alternative would be to carry *two* types for each pattern synonym: one to use in patterns, and one to use in expressions. Doing so would be perfectly sound and modular, but it is more complicated to explain, so we decided to stick with one single type for both the matcher (use as a pattern) and the builder (use as an expression).

In this example the matcher had a less-general type than the builder, but in explicitly-bidirectional pattern synonyms the reverse can be the case. Here is a contrived example:

$$\textbf{pattern}\ Q :: Ord\ a \Rightarrow a \to a \to (a, a)$$
$$\textbf{pattern}\ Q\ x\ y \leftarrow (x, y)$$
$$\quad \textbf{where}$$
$$\quad\quad Q\ x\ y \mid x \leqslant y \quad\ = (x, y)$$
$$\quad\quad\quad\ \mid otherwise = (y, x)$$

Here Q matches a pair; but when building a pair, Q puts the components into ascending order. Here matching needs no constraints, but building needs $(Ord\ a)$.

If no signature is supplied, we infer the type from the pattern *only*. So Q requires a signature, because the inferred signature would be $Q :: a \to b \to (a, b)$, which in this case is too general for the builder.

7. Implementation

We have fully implemented pattern synonyms in GHC, a state of the art compiler for Haskell. Over 70 third-party packages in Hackage already use the feature.

In this section we briefly sketch our implementation strategy. It has the following characteristics:

- It supports modular, separate compilation. In particular, pattern synonyms do not need to be expanded at their use sites, whether in patterns or in expressions.
- One might worry that the abstraction of pattern synonyms comes at an execution-time cost but in practice, for small definitions at least, the cost is zero.

7.1 Matchers and Builders

In the implementation, each pattern synonym P is associated with two ordinary function definitions: the *matcher* which is called when P is used in a pattern, and the *builder* which is called when P is used in an expression.

Consider the example from §6.2.

$$\textbf{data}\ RP\ a\ \textbf{where}$$
$$\quad MkRP :: (Eq\ a, Show\ b) \Rightarrow a \to b \to RP\ a$$
$$\textbf{pattern}\ P :: Num\ a \Rightarrow (Eq\ a, Show\ b) \Rightarrow b \to RP\ a$$
$$\textbf{pattern}\ P\ v = MkRP\ 3\ v$$

The builder for P, written $\$bP$, is straightforward:

$$\$bP :: (Num\ a, Eq\ a, Show\ b) \Rightarrow b \to RP\ a$$
$$\$bP\ v = MkRP\ 3\ v$$

The right hand side is taken directly from the definition of P, treating the pattern as an expression. (In an explicitly-bidirectional pattern synonym, the builder is defined in the **where** clause.)

The matcher for this example looks like this:

$$
\begin{aligned}
&\$mP :: \forall r\ a.Num\ a \\
&\quad \Rightarrow RP\ a \\
&\quad \rightarrow (\forall b.(Eq\ a, Show\ b) \Rightarrow b \rightarrow r) \\
&\quad \rightarrow r \\
&\quad \rightarrow r \\
&\$mP\ x\ sk\ fk = \mathbf{case}\ x\ \mathbf{of} \\
&\qquad\qquad MkRP\ 3\ v \rightarrow sk\ v \\
&\qquad\qquad \_ \qquad\quad \rightarrow fk
\end{aligned}
$$

The function matches its first argument x against P's defining pattern. If the match succeeds, the matcher passes the matched values (in this case just v) to the success continuation sk; if the match fails, the matcher returns the failure continuation fk.

The code for the matcher is very straightforward, but its type is interesting. In particular, notice that the success continuation is polymorphic in the $MkRP$'s existentially-bound type variables, and in its "provided" context $(Eq\ a, Show\ b)$. This is very natural: a successful match binds the existential variables and provided dictionaries.

7.2 Optimisation

It is a simple matter to adapt the pattern-matching compiler (a component of GHC's desugarer) to invoke the matcher $\$mP$ when P is encountered in a pattern. For example the definition

$$
\begin{aligned}
&f :: [RP\ Int] \rightarrow String \\
&f\ (P\ x : \_) = show\ x \\
&f\ ys \qquad\quad = show\ (length\ ys)
\end{aligned}
$$

desugars to

$$
\begin{aligned}
&f = \lambda ys \rightarrow \mathbf{let}\ fk = show\ (length\ ys)\ \mathbf{in} \\
&\qquad\quad \mathbf{case}\ ys\ \mathbf{of} \\
&\qquad\qquad (z : \_) \rightarrow \$mP\ z\ (\lambda x \rightarrow show\ x)\ fk \\
&\qquad\qquad [\,] \qquad\quad \rightarrow fk
\end{aligned}
$$

The empty-list of the outer pattern match and the failure continuation of P's matcher function both invoke the fk continuation which constitutes the second (and potentially subsequent) equations of f.

But this looks expensive! Invoking $\$mP$ means allocating a function closure for the success continuation. And that is necessary in general. But $\$mP$ is a perfectly ordinary function, and in many cases its definition is relatively small. GHC already has elaborate support for cross-module (and indeed cross-package) inlining, so if the implementation of $\$mP$ is indeed small, it will be inlined at the call site, and the result will be precisely as efficient as if you expanded the pattern-match at the usage site.

7.3 Unlifted Result Types

There is one wrinkle in this scheme, involving unboxed types (Peyton Jones and Launchbury 1991). Consider this Haskell program, where g is an expensive function with type $g :: Bool \rightarrow Int\#$:

$$
\begin{aligned}
&f :: RP\ Int \rightarrow Int\# \\
&f\ (MkRP\ 0\ \_) = 0\# \\
&f\ \_ \qquad\quad = g\ True
\end{aligned}
$$

We do not want to desugar this to

$$f = \lambda x \rightarrow \$mP\ x\ (\lambda v \rightarrow 0\#)\ (g\ True)$$

The trouble[8] is that since $(g\ True) :: Int\#$, GHC must use call-by-value for the third argument of $\$mP$, so the failure continuation will be expensively evaluated even when it is not going to be used. We solve this by adding a $Void\#$ argument to the failure continuation, so $\$mP$'s actual definition is

$$
\begin{aligned}
&\$mP :: \forall r\ a.Num\ a \\
&\quad \Rightarrow RP\ a \\
&\quad \rightarrow (\forall b.(Eq\ a, Show\ b) \Rightarrow b \rightarrow r) \\
&\quad \rightarrow (Void\# \rightarrow r) \\
&\quad \rightarrow r \\
&\$mP\ x\ sk\ fk = \mathbf{case}\ x\ \mathbf{of} \\
&\qquad\qquad MkRP\ 3\ v \rightarrow sk\ v \\
&\qquad\qquad \_ \qquad\quad \rightarrow fk\ void\#
\end{aligned}
$$

Adding the extra argument effectively gives us call-by-name (as desired) rather than call-by-value. There is no need for call-by-need, because the failure continuation can be invoked at most once.

If the pattern binds no variables then the success continuation may have the same problem, so in that case we add an extra $Void\#$ argument to the success continuation as well.

8. Evaluation

We studied the 9,705 packages uploaded to Hackage to evaluate the real-world usage of pattern synonyms. There are 17,644 unique definitions in 72 packages, but the large majority of these (16,826) are in machine generated code from libraries which wrap low level bindings. For example, the OpenGLRaw and gl packages define a total of 11,080 pattern synonyms in total.

This leaves 818 hand written definitions from 63 distinct packages which we can study. The most common form is implicitly bidirectional of which there are 506 definitions in 45 different packages. There are 137 explicitly bidirectional definitions in 11 packages. This number is greatly inflated by 3 packages defining 122 between them. This leaves 175 unidirectional pattern synonyms from 20 different packages.

On the other hand, pattern signatures are seldom used. There are just 29 pattern signatures on Hackage from 12 distinct packages. This is despite the accepted good practice of including type signatures for all top-level function definitions. Quite possibly, a reason for the dearth of pattern signatures is that the first release of GHC with pattern synonyms (7.8) did not include support for signatures, which became available only a year later. While we motivated the inclusion of pattern signatures as necessary for dealing with certain kinds of patterns, it can sometimes be hard to understand many pattern synonym definitions without signatures, especially when a pattern synonym is defined in terms of other pattern synonyms! In the latest version of GHC there is a warning which hopefully will encourage more authors to include pattern signatures, in parallel with the existing warning about missing signatures to ordinary functions.

The usage of pattern synonyms can be classified into three broad categories. By far the most prevalent usage is in libraries which provide Haskell bindings to libraries written in lower level languages, as mentioned above. The second class is using unidirectional pattern synonyms to extract commonly used parts of more complicated data types. The third (and least common) is for abstraction.

For example, pattern synonyms are used extensively by the OpenGLRaw[9] package which provides low level bindings to OpenGL. Many options in the library are passed to functions as $GLenum$s which is a type synonym for unsigned integers. This is

[8] Well-informed readers may also wonder about whether the type variable r can be instantiated by $Int\#$, an unboxed type. We use *representation polymorphism* (previously called levity polymorphism) for this, something that is already needed in GHC (Eisenberg 2015). We omit the details here.

[9] `https://hackage.haskell.org/package/OpenGLRaw`

difficult to represent natively as a sum type as the contexts which each constant can be used are not disjoint. Further, it is sometimes necessary to cast these constants to integers – in C, enums are just integers so casting is a no-op but in a strongly typed language these manual conversions become tedious.

Previous versions gave human readable names to each of these constants but this only allowed easy construction of values. Pattern matching could be achieved only through an indirect combination of pattern guards and equality tests. In recent versions, these have been replaced by pattern synonyms which also allow users to match on these values directly rather than relying on guards.

```
  { Earlier versions }
gl_MAJOR_VERSION :: GLenum
gl_MAJOR_VERSION = 0x821B
  { Recent versions }
pattern GL_MAJOR_VERSION :: GLenum
pattern GL_MAJOR_VERSION = 0x821B
```

Secondly, authors use pattern synonyms to improve readability. Authors define pattern synonyms in a local scope in order to improve the readability of just one or two functions. A real-world example can be found in §2.

Finally, the most surprising discovery to the authors was that explicitly bidirectional pattern synonyms are not yet widely used at all. There are only a handful of examples which can be found on Hackage. This is despite explicitly bidirectional synonyms being the most interesting (and powerful) because of the ability for more complicated abstractions. In non-trivial cases, the desired interface and internal representation will be quite different. As a result, inferring the builder from pattern is not possible. We are hoping that as pattern synonyms become more well-known (assisted by the publication of this paper!), Haskellers will find more good uses for the more powerful capabilities of this new feature.

Interestingly, a few packages use pattern synonyms to provide more efficient but unsafe representations of data types. The flat-maybe[10] package provides an alternative, user-land implementation of *Maybe* which uses only a single pointer. If the pointer points to a sentinel null value then the value is *Nothing*, otherwise the pointer points directly to the object we want.

```
newtype Maybe (a :: *) = Maybe Any

data Null = Null

null :: Null
null = Null

pattern Nothing :: Maybe a
pattern Nothing ← (isNothing# → 1#) where
  Nothing = (unsafeCoerce # null :: Maybe a)

pattern Just :: a → Maybe a
pattern Just a ← (isJust# → (#0#, a#)) where
  Just (a :: a) = (unsafeCoerce # a :: Maybe a)
```

isNothing# and *isJust#* are implemented using unsafe compiler primitives but pattern synonyms provide a safe interface. structs[11] is another library to experiment with this idea.

Several packages surveyed make use of unidirectional pattern synonyms to allow complex deconstructing of abstract types but then separately define builders rather than using explicitly bidirectional synonyms. We hope that these library authors will begin to use these more advanced features for interesting applications. Generally library authors wish to support three major versions of GHC which leads to slow widespread adoption of new features.

[10] `https://hackage.haskell.org/package/flat-maybe`

[11] `https://hackage.haskell.org/package/structs`

9. Related Work

The most closely related to our work is *abstract value constructors* as described by Aitken and Reppy (1992). They similarly propose to give names to patterns which can then be used in both patterns and expressions like our implicitly bidirectional pattern synonyms. We extend their work by introducing unidirectional and explicitly bidirectional pattern synonyms as well as signatures. Further to this, because of view patterns, this simple implementation is significantly more powerful than if we were to implement it in a language such as SML where computation can not be performed in patterns.

Views are also very closely related. First described by Wadler (1987) in his proposal, a user provides inverse functions **in** and **out** which form an isomorphism between the internal representation and user-visible representation. As we have seen, pattern synonyms can be used to implement views but are more general. Recently, views have become popular in dependently typed programming languages where they are used in order to prove properties about abstract data types (McBride and McKinna 2004).

Burton et al. (1996) proposed an extension to implement views in Haskell. At the time of the proposal, view patterns had not been implemented; it was thus not possible to perform computation in patterns. Their proposal also explicitly disallowed use in expression contexts which is at odds with the goal of abstraction. Okasaki (1998) also described a very similar proposal of views for Standard ML with some additional discussion about how they should behave relative to state and a module system.

There are also several modern implementations. With F#'s active patterns (Syme et al. 2007) users can either define *total active patterns* which provide total decomposition akin to views or *partial active patterns* which are more similar to pattern synonyms. Further to this, active patterns can also be parameterised. In this case, the patterns are defined by functions of type $a_1 \to ... \to a_n \to s \to Maybe\ t$ and at the use site a user must first apply the pattern to values of type $a_1, ..., a_n$. For example, it is possible to define a single pattern which takes a regular expression as an argument before using the regular expression in order to check whether a string matches. Active patterns can be used only in patterns.

Simple pattern synonyms are implemented in Agda (The Agda Team 2016). In our terminology, it is possible to define only implicitly bidirectional pattern synonyms. The implementation technique is also much simpler – after renaming, the definition is directly substituted into the abstract syntax tree. Patterns thus do not have types and cannot enforce abstraction.

SHE (McBride 2009) (the Strathcylde Haskell Enhancement) implements pattern synonyms for Haskell as a preprocessor. Users can choose to define either unidirectional or implicitly bidirectional pattern synonyms from a restricted pattern syntax (the patterns which look the same in both contexts). Before parsing, SHE performs textual substitution to desugar all occurrences of pattern synonyms in the file.

In Scala, users can define special objects called *extractors* (Emir et al. 2007) by specifying `apply` and `unapply` functions which specify how the object should behave in expressions and patterns respectively. Given an extractor `T`, in expressions Scala desugars `T(x)` to `T.apply(x)` whilst in patterns the `unapply` method is called in order to determine whether a value matches. Extractors share a likeness to explicitly bidirectional pattern synonyms however it is not necessary to provide both methods – one can choose to omit either.

Idris also has a simple form of pattern synonyms due to the support for extensible syntax (Brady 2015). Users can define custom syntactic forms which can also appear in patterns and are inserted by template substitution.

10. Future Work

10.1 Associated Pattern Synonyms

In our discussion of polymorphic pattern synonyms in §2.3 we noted that one could define a type class in order to characterise the ways to build and match on a data type and then define a set of pattern synonyms which used just those methods which could be implemented in many different ways.

To take this idea a step further, instead of this indirect implementation method we could instead allow pattern synonyms to be included in the definition of a class.

```
class ListLike f where
  pattern Nil :: f a
  pattern Cons :: a → f a → f a
instance ListLike [] where
  pattern Nil = []
  pattern Cons x xs = (x : xs)
```

This is mainly a syntactic nicety; if we extended bundling (as described in §4.7.2) to also allow pattern synonyms to be bundled with type classes then we could recover much of the power of this proposal with very little effort.

In fact, for complex representations, it is still necessary to define complex projection and injection functions. As a result, using associated pattern synonyms is perhaps *more* work than the approach demonstrated previously as a user must first define these complex functions and then add on annoying boiler plate in order to define the pattern synonym. One way around this might be to allow default definitions to be included in the class definition but it is not clear what this gains from the simpler extension to bundling.

There are also syntactic problems to overcome. Specifying the signature for a pattern synonym does not give enough indication whether one means a unidirectional or bidirectional definition nor a prefix or record pattern synonym. It is important that each class implements the same kind of pattern synonym as otherwise the choice of instance determines in which contexts one can use the associated pattern synonym.

10.2 Exhaustiveness Checking

Despite being a very common way in which to write functions, it is all too easy to write non-total functions when pattern matching. GHC includes a very robust exhaustiveness checker (Karachalias et al. 2015) in order to check that functions defined in this way are total. Due to the complexity of Haskell's pattern language, there are already a number of cases in which it is undecidable whether a pattern match is exhaustive (e.g., with view patterns).

Pattern synonyms provide an interesting addition to this challenge. As they are defined in terms of patterns, one could simply look through a pattern synonym in order to attempt to verify whether the pattern matches are exhaustive. However, this solution is at odds with abstraction. It is undesirable to expose the internal representation through exhaustiveness warnings.

A set of pattern synonyms may be exhaustive even if the underlying patterns are not. As one such example, a user might want to represent command line flags by using strings. In order to hide this representation, she provides an abstract type *Flag* with some bidirectional pattern synonyms:

```
module Flags (Flag (AddOne, MinusOne)) where
newtype Flag = Flag String
pattern AddOne   = Flag "add-one"
pattern MinusOne = Flag "minus-one"
```

Clients of this module can construct *Flag*s via only these synonyms, and thus these synonyms cover the entire space of possible *Flag*s. Enabling the exhaustiveness checker to be aware of these seems to require user intervention, but it would be convenient to be able to do so. Then in order to perform an exhaustive match, a user would need to match on each pattern synonym in the group.

Conclusion Pattern synonyms extend the Haskell language with a useful new way of abstraction; the quick uptake by third-party package maintainers validates this view. Furthermore, since abstraction in Haskell requires types, introducing this feature led to the intricacies of pattern types, previously only implicitly defined and loosely understood.

References

W. E. Aitken and J. H. Reppy. Abstract value constructors. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 1–11, 1992.

E. Brady. Idris documentation: Syntax extensions. `http://docs.idris-lang.org/en/latest/tutorial/syntax.html`, 2015.

W. Burton, E. Meijer, P. Sansom, S. Thompson, and P. Wadler. A (sic) extension to Haskell 1.3 for views. Mailing List, October 1996.

R. A. Eisenberg. An overabundance of equality: Implementing kind equalities into Haskell. Technical Report MS-CIS-15-10, University of Pennsylvania, 2015. URL `http://www.cis.upenn.edu/~eir/papers/2015/equalities/equalities.pdf`.

B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *Proceedings of the 21st European conference on Object-Oriented Programming*, pages 273–298. Springer-Verlag, 2007.

M. Jaskelioff and E. Rivas. Functional pearl: a smart view on datatypes. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 355–361. ACM, 2015.

G. Karachalias, T. Schrijvers, D. Vytiniotis, and S. P. Jones. GADTs meet their match. In *International Conference on Functional Programming, ICFP*, volume 15, 2015.

S. Marlow et al. Haskell 2010 language report. *Available online http://www.haskell. org/(May 2011)*, 2010.

C. McBride. The Strathclyde Haskell Enhancement. `https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/`, 2009. Accessed: 2016-03-23.

C. McBride and J. McKinna. The view from the left. *Journal of functional programming*, 14(01):69–111, 2004.

C. Okasaki. Views for Standard ML. In *SIGPLAN Workshop on ML*, pages 14–23, 1998.

S. Peyton Jones and J. Launchbury. Unboxed values as first class citizens. In R. Hughes, editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523, pages 636–666, Boston, 1991.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM.

T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *ACM Sigplan Notices*, volume 44, pages 341–352. ACM, 2009.

D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *ACM SIGPLAN Notices*, volume 42, pages 29–40. ACM, 2007.

The Agda Team. Release notes for Agda 2 version 2.3.2. `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Version-2-3-2`, 2016. Accessed: 2016-03-23.

P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 307–313. ACM, 1987.

A. Formalisation

To formalise the typing of pattern synonyms, we need a model language that has all the features that are interesting in their interplay with pattern matching. The model language formalised here has the following features:

- Nested pattern matching on data type values
- Parametric and typeclass-based polymorphism: types are of the form $\forall\overline{\alpha}.\Theta \Rightarrow \tau$, where $\overline{\alpha}$ are the bound type variables, and Θ contains typeclass and equality constraints on them. Polymorphism is introduced by **let** bindings and top-level definitions (lambdas are monomorphic).
- GADT-like data constructors: the type of data constructors can contain typeclass and equality constraints and existentially-bound type variables: $K : \forall\overline{\alpha}\ \overline{\beta}.\Theta \Rightarrow \overline{\tau} \to T\ \overline{\alpha}$ is a valid data constructor for the data type $T\ \overline{\alpha}$. As described in §3.3, we can recover the full power of GADTs just with equality constraints on data constructors.
- View patterns: pattern matching can do arbitrary computation by applying a function on the scrutinee and then pattern matching on the result. In our presentation here, there is no flow of information from preceding pattern matches into the view expression, unlike in Haskell. This difference is immaterial to our topic.

Figures 4 and 5 describe the syntax and inference rules of our type system. Three kinds of contexts are used:

- The context Γ of (term) variables, mapping variable names to (potentially polymorphic) types.
- Σ collects the instance constraints
- Ψ is the set of rigid (existentially-bound) type variables that cannot be abstracted over.

Other contextual information is taken as axiomatic:

- Data types and data constructors are assumed to be given *a priori*
- There is no facility to introduce new typeclasses. Overloaded functions are assumed to be given in the initial Γ.
- Types are assumed to be well-formed and valid: type variable scoping is not tracked and no kind checking is done.

To add pattern synonyms to this formalisation as a proper abstraction of patterns (i.e., adhering to the *Typing Principle*), they must be characterised by a type that allows pattern typing statements of the form

$$\frac{P :? \qquad \Gamma_0, (\Sigma_0, ?) \vdash pat_i : \tau_i \rightsquigarrow \Gamma_i, \Sigma_i, \Psi_i}{\Gamma_0, \Sigma_0 \vdash P\ pat_1 \ldots pat_n :? \rightsquigarrow \overline{\Gamma}, (\overline{\Sigma}, ?), (\overline{\Psi}, ?)}$$

where $P\ pat_1 \ldots pat_n$ is a fully-applied pattern synonym. As the above shows, the requirement on the characterisation of a pattern synonym is to determine:

- The extension (if any) to the instance context Σ_0 in which sub-patterns and the right-hand side are checked
- The extension (if any) to the set of rigid type variables
- Typing of the pattern synonym itself, i.e. its requirements on the instance context Σ_0 and the relation between sub-pattern types τ_i and the pattern synonym application's type

By looking at each possible way a pattern synonym can be defined (as a variable, as a wildcard pattern, as a data constructor pattern of some arity, or as a view pattern), we can see that to observe the *Typing Principle*, a pattern synonym type for a pattern synonym $P\ p_1 \ldots p_n$ must account for:

$expr\ e, f$	$::=$	con	Constructor
	$\mid$	$expr\ expr$	Function application
	$\mid$	var	Variable occurrence
	$\mid$	'λ' var '$\mapsto$' $expr$	Lambda abstraction
	$\mid$	'**case**' $expr$ '**of**' alt	Branching
	$\mid$	'**let**' var '$=$' $expr$ '**in**' $expr$	Variable binding
alt	$::=$	pat '$\mapsto$' $expr$	Alternative

Figure 4. Syntax of expression

- The type schema of the scrutinee $\forall\overline{\alpha}.\Theta_{\text{req}} \Rightarrow \tau$. The required instance context Θ_{req} is used for view patterns in instantiating polymorphic view functions;
- The set of newly-introduced rigid type variables $\overline{\beta}$, used for data constructor patterns,
- The types of the arguments $p_1 : \tau_1, \ldots, p_n : \tau_n$, where τ_i can contain free variables from $\overline{\alpha}$ and $\overline{\beta}$,
- The instance context extension Θ_{prov} provided by data constructor patterns

leading to the pattern synonym type and the typing rule shown on figure 6. Indeed, this corresponds to the pattern signature described in §6.

$\boxed{\Gamma, \Sigma \vdash e : \tau}$

$$\frac{K : \forall \overline{\alpha}.\Theta \Rightarrow \tau_0 \qquad \Sigma \vdash \Theta[\overline{\tau}/\overline{\alpha}]}{\Gamma, \Sigma \vdash K : \tau_0[\overline{\tau}/\overline{\alpha}]} \qquad \frac{(x : \forall \overline{\alpha}.\Theta \Rightarrow \tau_0) \in \Gamma \qquad \Sigma \vdash \Theta[\overline{\tau}/\overline{\alpha}]}{\Gamma, \Sigma \vdash x : \tau_0[\overline{\tau}/\overline{\alpha}]}$$

$$\frac{\Gamma, \Sigma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma, \Sigma \vdash e : \tau_1}{\Gamma, \Sigma \vdash f\ e : \tau_2} \qquad \frac{\Gamma, x : \tau_1, \Sigma \vdash e : \tau_2}{\Gamma, \Sigma \vdash \lambda x \mapsto e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma, \Sigma \vdash e_0 : \tau_0 \qquad \Gamma, \Sigma \vdash pat_i \mapsto e_i : \tau_0 \mapsto \tau}{\Gamma, \Sigma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ \overline{pat_i \mapsto e_i} : \tau}$$

$$\frac{\Gamma, x : \tau_0, (\Sigma, \Theta) \vdash e_0 : \tau_0 \qquad FV(\Gamma) \cap \overline{\alpha} = \emptyset \qquad FV(\Theta) \subseteq \overline{\alpha} \qquad \Gamma, x : \forall \overline{\alpha}.\Theta \Rightarrow \tau_0,\ \Sigma \vdash E : \tau}{\Gamma, \Sigma \vdash \mathbf{let}\ x = e_0\ \mathbf{in}\ e : \tau}$$

(a) Typing rules of expression

$\boxed{\Gamma, \Sigma \vdash alt : \tau_0 \mapsto \tau}$

$$\frac{\Gamma, \Sigma \vdash pat : \tau_0 \rightsquigarrow \Gamma', \Sigma', \Psi \qquad \Gamma, \Gamma', \Sigma, \Sigma' \vdash e : \tau \qquad \Psi \cap FV(\tau) = \emptyset}{\Gamma, \Sigma \vdash pat \mapsto e : \tau_0 \mapsto \tau}$$

(b) Typing rules of alternatives

$\boxed{\Gamma_0, \Sigma_0 \vdash pat : \tau \rightsquigarrow \Gamma, \Sigma, \Psi}$

$$\frac{}{\Gamma_0, \Sigma_0 \vdash x : \tau \rightsquigarrow x : \tau, \emptyset, \emptyset} \qquad \frac{}{\Gamma_0, \Sigma_0 \vdash \_ : \tau \rightsquigarrow \emptyset, \emptyset, \emptyset}$$

$$\frac{K : \forall \overline{\alpha}\ \overline{\beta}.\Theta \Rightarrow \tau_1 \ldots \tau_n \to T\ \overline{\alpha} \qquad \Gamma_0, (\Sigma_0, \Theta[\overline{\tau}'/\overline{\alpha}]) \vdash pat_i : \tau_i[\overline{\tau}'/\overline{\alpha}] \rightsquigarrow \Gamma_i, \Sigma_i, \Psi_i}{\Gamma_0, \Sigma_0 \vdash K\ pat_1 \ldots pat_n : T\ \overline{\tau}' \rightsquigarrow \overline{\Gamma}, (\overline{\Sigma}, \Theta[\overline{\tau}'/\overline{\alpha}]), (\overline{\Psi}, \overline{\beta})}$$

$$\frac{\Gamma_0, \Sigma_0 \vdash e : \tau \to \tau' \qquad \Gamma_0, \Sigma_0 \vdash pat : \tau' \rightsquigarrow \Gamma, \Sigma, \Psi}{\Gamma_0, \Sigma_0 \vdash (e \to pat) : \tau \rightsquigarrow \Gamma, \Sigma, \Psi}$$

(c) Typing rules of patterns

Figure 5. Syntax and typing rules

$$\begin{array}{rcll} pat & ::= & \ldots & \text{(see figure 1)} \\ & | & P\ pat_1 \ldots pat_n & \text{Pattern synonym} \end{array}$$

$$\frac{P : \forall \overline{\alpha}.\Theta_{\text{req}} \Rightarrow \forall \overline{\beta}.\Theta_{\text{prov}} \Rightarrow \tau_1 \to \ldots \to \tau_n \to \tau \qquad \Sigma_0 \vdash \Theta_{\text{req}}[\overline{\tau}'/\overline{\alpha}] \qquad \Gamma_0, (\Sigma_0, \Theta_{\text{prov}}[\overline{\tau}'/\overline{\alpha}]) \vdash pat_i : \tau_i[\overline{\tau}'/\overline{\alpha}] \rightsquigarrow \Gamma_i, \Sigma_i, \Psi_i}{\Gamma_0, \Sigma_0 \vdash P\ pat_1 \ldots pat_n : \tau[\overline{\tau}'/\overline{\alpha}] \rightsquigarrow \overline{\Gamma}, (\overline{\Sigma}, \Theta_{\text{prov}}[\overline{\tau}'/\overline{\alpha}]), (\overline{\Psi}, \overline{\beta})}$$

Figure 6. New syntax and pattern typing rule for pattern synonyms