Haskell Type Constraints Unleashed: Companion Report

Dominic Orchard
Tom Schrijvers

Report CW 574, January 2010



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Haskell Type Constraints Unleashed: Companion Report

Dominic Orchard
Tom Schrijvers

Report CW 574, January 2010

Department of Computer Science, K.U.Leuven

Abstract

The popular Glasgow Haskell Compiler extends the Haskell 98 type system with several powerful features, leading to an expressive language of type terms. In contrast, constraints over types have received much less attention, creating an imbalance in the expressivity of the type system. In this paper, we rectify the imbalance, transferring familiar type-level constructs, *synonyms* and *families*, to the language of constraints, providing a symmetrical set of features at the type-level and constraint-level. We introduce *constraint synonyms* and *constraint families*, and illustrate their increased expressivity for improving the utility of polymorphic EDSLs in Haskell, amongst other examples. We provide a discussion of the semantics of the new features relative to existing type system features and similar proposals, including details of termination.

Keywords : Haskell, GHC, type system, type constraints, indexed types.

Haskell Type Constraints Unleashed

Dominic Orchard[1] and Tom Schrijvers[2⋆]

[1] Computer Laboratory, University of Cambridge
dominic.orchard@cl.cam.ac.uk
[2] Katholieke Universiteit Leuven
tom.schrijvers@cs.kuleuven.be

Abstract. The popular Glasgow Haskell Compiler extends the Haskell 98 type system with several powerful features, leading to an expressive language of type terms. In contrast, constraints over types have received much less attention, creating an imbalance in the expressivity of the type system. In this paper, we rectify the imbalance, transferring familiar type-level constructs, *synonyms* and *families*, to the language of constraints, providing a symmetrical set of features at the type-level and constraint-level. We introduce *constraint synonyms* and *constraint families*, and illustrate their increased expressivity for improving the utility of polymorphic EDSLs in Haskell, amongst other examples. We provide a discussion of the semantics of the new features relative to existing type system features and similar proposals, including details of termination.

1 Introduction

The functional programming language Haskell has a rich set of type system features, as described by the Haskell 98 standard [1], which the Glasgow Haskell Compiler (GHC) has extended considerably[3]. Types in Haskell consist of two parts, the *type* term τ and *constraint* term C, forming a *qualified type* $C \Rightarrow \tau$. The syntactic position to the left of $\Rightarrow$ is known as the *context* of a type, which may be empty.

The majority of GHC's type system features extend the language of type terms τ. The type term language includes, from the Haskell 98 standard: *algebraic data types* and *type synonyms*, and added by GHC: *generalised algebraic data types* (GADTs), *type synonym families* [2], and *data type families* [3]. In contrast, the language of constraints C has received little attention, consisting of only *type classes* (from Haskell 98) and GHC's *equality constraints* [4].

Recently, Haskell has been recognised as a good host for polymorphic *embedded domain-specific languages* (EDSLs) [5]. However, limitations of the relatively inexpressive constraint language have become even more apparent with this recent wave of EDSLs, that exploit the possibilities of the rich type-term language,

⋆ Post-doctoral researcher of the Fund for Scientific Research - Flanders.

[3] From now on, reference to Haskell means the Haskell language as accepted by GHC, as of December 2009.

but find the constraint language lacking (see example problems in Section 2). We rectify this imbalance, expanding Haskell's capacity for expressing EDSLs.

We introduce two new type system features, *constraint synonyms* (Section 4) and *constraint synonym families* (sometimes abbreviated to *constraint families*) (Section 5) which tackle some of the problems faced with the inflexible constraint language. We discuss the syntax and semantics of the new features, paying particular attention to the termination conditions of constraint families.

Our new features are a natural extrapolation of GHC's type-term constructions to the constraint-term language i.e. the new features are derived by analogy, as opposed to being added in an *ad-hoc* manner (see Section 3). The new features do not extend the power of Haskell, but can be encoded in the existing language. An encoding schema and implementation is discussed in Section 6. Section 7 provides a discussion of existing proposals for solutions to the inflexible constraint language, such as *class aliases* [6, 7] and *class families* [8].

2 Problems

We see two kinds of problem with Haskell's current constraint-term language: the first related to the naming of constraints, the second related to *type-indexing*, or *type-dependency*, of constraints.

2.1 Naming Problems

Haskell does not provide a renaming feature for constraints or conjunctions of multiple constraints. There are several undesirable situations which arise due to the lack of such renaming features.

Tiresome Repetition of Long Constraints Generalised frameworks and libraries factor out instance-specific functionality and algorithms as (higher-order) parameters. Type classes provide considerable convenience in making these parameters implicit. Related functionality can be grouped in a single type class, but in the case of orthogonal design choices, multiple type classes may be involved. Consequently, highly flexible functions accumulate a large number of constraints in their signature, yielding unwieldy, large, and hard to understand constraint terms.

Consider, for instance, the signature of the `eval` function in the Monadic Constraint Programming framework [9]:

```
eval :: (Solver s, Queue q, Transformer t,
        Elem q ~ (Label s, Tree s a, TreeState t), ForSolver t ~ s) => ...
```

where the first three type class constraints capture different implicit parameters and the last two equality constraints enforce consistency. Such a long constraint term is cumbersome to read and write for the programmer.

Cumbersome Refactoring Type classes can modularise the design of libraries, where independently useful classes provide more generalised behaviour than their subclasses. An existing type class might be refactored by decomposition into several smaller, more general, independent subclasses. For example, the pervasive `Num` class could be divided into `Additive`, `Multiplicative`, and `FromInteger` classes. Unfortunately, this decomposition means any program with explicit `Num` constraints and/or type instances of the `Num` class must be rewritten, rewriting explicit constraints and refactoring instances into several smaller instances.

2.2 Uniformity Problems

Type classes impose a uniform structure on all instances; all methods have type signatures of a fixed shape. However, associated type families allow variation in the shape of signatures by making them (partly) dependent on the instance type. No such flexibility is available with constraint terms, thus type classes impose uniform constraints on all instances, considerably restricting possible instances.

Constrained Functors The `Functor` type class is a good example of a larger class of constructor type classes, including also monads.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Members of `Functor` implement the `fmap` function, which lifts a function to operate over a parameterised data type. The context of the `fmap` function is empty – there are no constraints – which is sufficient for many data types, such as lists. Other data types may impose constraints on the parameter types `a`, `b`. For instance, the `Set` data type from Haskell's standard library provides a map function over `Set` with type:

$$Set.map :: (Ord\ a, Ord\ b) \Rightarrow (a \to b) \to Set\ a \to Set\ b$$

which almost matches the signature of `fmap`, except its non-empty context `(Ord a, Ord b)`. To generalise the `Functor` type class, we would like the context of `fmap` to depend on the type `f`.

Polymorphic Final EDSLs A similar issue has recently appeared[4] in the development of final tagless EDSLs [10] of a polymorphic nature. Consider the following EDSL of simple polymorphic expressions of constants and addition:

```
class Expr sem where
  constant :: a -> sem a
  add      :: sem a -> sem a -> sem a
```

where `add (constant 1) (constant 2)` would denote the integer expression $1 + 2$ while `add (constant 1.0) (constant 2.0)` would denote the real expression $1.0 + 2.0$. We would like to implement an evaluation semantics `E`:

[4] `http://www.haskell.org/pipermail/haskell-cafe/2009-September/066670.html`

```
data E a = E {eval :: a}

instance Expr E where
  constant c  = E c
  add e1 e2   = E $ eval e1 + eval e2
```

However, this code fails to type check because addition `(+)` requires the constraint `Num a`. We could solve this problem by adding the `Num a` constraint to the method signature of `add` in the class declaration. However, other useful semantics for this polymorphic EDSL may require their own constraints, e.g. `Show` for printing, `Read` for parsing, etc.

Adding more and more constraints to the methods of `Expr` prevents modular reuse and narrows the number of possible monomorphic instances of the polymorphic EDSL. In other words, the polymorphic nature of the EDSL becomes almost pointless.

Polymorphic HOAS-based EDSL A similar problem arises in the polymorphic EDSL for constraint models `Tree s a` of the Monadic Constraint Programming framework [9]. This data type has a constrained (i.e. GADT) *higher-order abstract syntax* (HOAS) constructor:

```
data Tree s a where
        NewVar  :: Term s t => (t -> Tree s a) -> Tree s a
        ...
```

where `Tree s t` expresses that `t` is one of (possibly) several term types supported by the constraint solver of type `s`. The `Term` class provides one method:

```
class (Solver solver) => Term solver term where newvar :: solver term
```

However, for specific solvers, we would like to capture that additional methods are available. Unfortunately, we cannot refine the `NewVar` constructor's constraint based on the particular solver type `s`.

3 Our Approach

We introduce two new constraint term features, *constraint synonyms* and *constraint synonym families*, analogous to existing type term features.

We classify GHC's existing type system features into two groups: type terms and constraint terms. Within these groups there are two further levels of subdivision. The second division differentiates between *family* terms (indexed-sets) and *constant* terms (nullary families). The third division differentiates whether families or constants are synonym definitions or *generative* definitions – that is, definitions that generate unique, non-substitutable constructors. These three levels of division essentially give a three-dimensional design space which is partially inhabited by GHC's existing features.

Fig.1 summarises existing GHC features and names the missing features in the design space (in bold). In this paper we consider *constraint synonyms* (Section 4) and *constraint synonym families* (Section 5). The third feature missing

		types	*constraints*
constants	*generative*	Data types[4] **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
	synonym	Type synonyms **type** $T\ \bar{a}\ =\ \tau$	**Constraint synonyms** **constraint** $K\ \bar{a}\ =\ C$
families	*generative*	Data type families **data family** $T\ \bar{a}$ **data instance** $T\ \bar{\tau}\ =\ \ldots$	**Class families** not considered
	synonym	Type synonym families **type family** $T\ \bar{a}$ **type instance** $T\ \bar{\tau}\ =\ \tau$	**Constraint synonym families** **constraint family** $K\ \bar{a}$ **constraint instance** $K\ \bar{\tau}\ =\ C$

Fig. 1. Type-level features extrapolated to the sparse constraint-level.

from the constraint language, *class families*, has been informally discussed in the Haskell community (see Section 7), but we do not know of any problems which cannot be solved more elegantly with constraint synonym families.

Fig.1 also introduces the syntax of constraint synonyms and constraint families, derived systematically from the corresponding type-side syntax.

3.1 Preliminaries

We adopt the following syntax for constraint terms:

$$C, D\ ::=\ () \mid (C_1, \ldots, C_n) \mid \tau_1 \sim \tau_2 \mid K\ \bar{\tau}$$

The first three of these constraint forms are built into the language:

- **True** - The empty constraint `()` denotes truth. Existing Haskell syntax does not provide an explicit notation for this concept; the whole constraint context $C \Rightarrow$ is simply omitted. For both new features that we introduce, the explicit notation becomes necessary.
- **Conjunction** - The tuple $(C_1, \ldots, C_n)$ denotes the n-way conjunction $\bigwedge_{i=1}^{n} C_i$. Conjunction is associative and commutative, but the tuple notation is not.
- **Equality** - The notation $\tau_1\ \sim\ \tau_2$ denotes equality of the types τ_1 and τ_2.

The fourth form of constraints, $K\ \bar{\tau}$, are user-defined, where K is the name of a constraint constructor and $\bar{\tau}$ is its type parameters. A user-defined constructor of arity n must be fully applied to exactly n type arguments. Currently, Haskell provides only one form of user-defined constraints: type class constraints. Our new features add two new forms for constraint synonyms and constraint families.

Constraints figure in the static semantics of Haskell essentially in the (Var) rule, for typing a variable expression x:

$$(\textsc{Var})\ \frac{(x : \forall \bar{a}.D \Rightarrow \tau) \in \Gamma \quad \theta = [\bar{\tau}/\bar{a}] \quad C \models \theta(D)}{C; \Gamma \vdash x : \theta(\tau)}$$

[4] The term "data types" refers to GADTs, of which ADTs are a special case.

a). *Existing Constraint Features:*

$$(\textsc{True})\ \frac{}{C \models ()} \qquad (\textsc{Conj})\ \frac{C \models D_1 \quad \cdots \quad C \models D_n}{C \models (D_1, \ldots, D_n)}$$

$$(\textsc{Given})\ \frac{}{C \models C} \qquad (\textsc{Decomp})\ \frac{C_i \models D}{(\ldots, C_i, \ldots) \models D}$$

$$(\textsc{Inst})\ \frac{\texttt{instance}\ D\ \texttt{=>}\ K\ \bar{\tau} \qquad C \models \theta(D)}{C \models \theta(K\ \bar{\tau})} \qquad (\textsc{Super})\ \frac{\texttt{class}\ C\ \texttt{=>}\ K\ \bar{a} \qquad \theta(C) \models D}{\theta(K\ \bar{a}) \models D}$$

b). *Constraint Synonyms:*

$$(\text{W-}\textsc{Syn})\ \frac{\texttt{constraint}\ K\ \bar{a}\ \texttt{=}\ D \qquad C \models \theta(D)}{C \models \theta(K\ \bar{a})} \qquad (\text{G-}\textsc{Syn})\ \frac{\texttt{constraint}\ K\ \bar{a}\ \texttt{=}\ D \qquad \theta(D) \models C}{\theta(K\ \bar{a}) \models C}$$

c). *Constraint Synonym Families:* (`constraint` *keyword omitted for compactness*)

$$(\text{W-}\textsc{Syn-Fam})\ \frac{\texttt{family}\ K\ \bar{a} \qquad \texttt{instance}\ K\ \bar{\tau}\ \texttt{=}\ D \qquad C \models \theta(D)}{C \models \theta(K\ \bar{\tau})} \qquad (\text{G-}\textsc{Syn-Fam})\ \frac{\texttt{family}\ K\ \bar{a} \qquad \texttt{instance}\ K\ \bar{\tau}\ \texttt{=}\ D \qquad \theta(D) \models C}{\theta(K\ \bar{\tau}) \models C}$$

Fig. 2. Constraint Implication

which states that expression x has type $\theta(\tau)$ (where θ is a substitution of type variables for types) with respect to environment Γ and given constraints C, if:

1. x has a more general type $\forall \bar{a}.D \Rightarrow \tau$ in the environment, and
2. constraint $\theta(D)$ is entailed by given constraints C, denoted $C \models \theta(D)$.

The entailment relation $\models$ defines the static meaning of constraints. **Fig.2a** defines $\models$ for the built-in constraint constructors, ignoring equality constraints. We refer to [4, Fig. 3] for the semantics of type equalities, which involve minor adjustments to the form of the judgements that are irrelevant to our purposes.

4 Constraint Synonyms

Haskell's *type synonyms* provide a simple mechanism for specifying renamings of types which can be rewritten (desugarded) prior to type checking. Rewriting does not change the meaning of a program. *Constraint synonyms* provide the same functionality as type synonyms but on constraint terms, with syntax:

$$\texttt{constraint}\ K\ \bar{a} = C$$

where $ftv(C)$ is the set of (free) type variables in C and $ftv(C) \subseteq \bar{a}$.

The two rules of **Fig.2b** extend constraint entailment to constraint synonyms in the obvious way. The (W-Syn) rule defines entailment of a *wanted* synonym constraint $K\ \bar{a}$, that must be satisfied by existing, *given* constraints C; the (G-Syn) rule defines entailment of a wanted constraint C, which is satisfied by a given constraint synonym $K\ \bar{a}$.

4.1 Examples

Revisiting the problems of Section 2.1, we see the advantage of constraint synonyms. Firstly, a large constraint term can be conveniently abbreviated:

```
constraint Eval s q t a =
       (Solver s, Queue q, Transformer t,
        Elem q ~ (Label s, Tree s a, TreeState t), ForSolver t ~ s)

eval :: Eval s q t a => ...
```

Secondly, for decomposition of monolithic classes, such as `Num`, into a number of smaller, independent, more general classes, the original constraint constructor can be preserved as the synonym of a conjunction of its subclasses e.g.

```
constraint Num a = (Additive a, Multiplicative a, FromInteger a)
```

Thus, existing explicit `Num` constraints do not need rewriting. However, `Num` instances must be rewritten as instances of the subclasses `Additive`, `Multiplicative`, etc. Proposed *class aliases* [6] do not have this problem (see Section 7).

4.2 Termination

Without restrictions, ill-founded constraint synonyms may arise, such as:

```
constraint K1 a = K1 a
constraint K2 a = (K2 [a], Eq a)
```

Neither of these is a synonym of any well-founded synonym-free constraint term. Hence, we enforce that constraint synonym definitions form a terminating rewrite system, when interpreted as left-to-right rewrite rules, in the same way as the Haskell 98 rule for type synonyms. Therefore the *call graph* of synonym definitions must be acyclic: a synonym may not appear in the right-hand side of its definition, or transitively in the right-hand side of any synonym mentioned.

5 Constraint Synonym Families

Type families allow types to depend upon, or be *indexed* by, other types [2–4]. Constraint synonym families extend this capability to the constraint terms of a type, giving type-indexed constraints. A new constraint family K is defined by:

`constraint family` $K\ \bar{a}$

The definition of families is *open*, allowing instances to be defined in other modules that are separately compiled. An instance is defined by:

$$\texttt{constraint instance}\ K\ \bar{\tau} = C$$

where the arity of instances must match the arity of the family declaration, and $ftv(C) \subseteq ftv(\bar{\tau})$.

Families can also be *associated* to a type class by declaration inside a class body. This nesting requires that all n parameters of the class are repeated as the first n parameters of the associated family, and that all additional parameters are distinct free type variables. This ensures that there is exactly one constraint family instance for every host type class instance. Note, that this differs from associated type families and data families, whose parameters must be exactly those of the parent class. Possible additional named parameters, further to the class parameters, are required as constraints on the right-hand side of a family instance must be fully applied, thus a point-free style cannot be admitted.

The `family` and `instance` keywords are elided when a family is associated with a class:

$\texttt{class}\ K\ \bar{a}\ \texttt{where}$
$\quad \texttt{constraint}\ KF\ \bar{a}\ \bar{b}$

$\texttt{instance}\ K\ \bar{\tau}\ \texttt{where}$
$\quad \texttt{constraint}\ KF\ \bar{\tau}\ \bar{b} = C$

Constraint entailment is extended to constraint synonym families where entailment of instances proceeds in the same way as constraint synonym entailment (see **Fig.2c**, rules (W-SYN-FAM) and (G-SYN-FAM)).

5.1 Examples

The following examples show how the problems of Section 2.2 can be solved with constraint synonym families. They also address two additional aspects: *default values* and the lack of need for *explicit implication constraints*.

Constrained Functors The `Functor` type class is generalised to impose a constraint, `Inv`, indexed by the functor type.

```
class Functor f where
  constraint Inv f e
  fmap :: (Inv f a, Inv f b) => (a -> b) -> f a -> f b
```

Both lists and sets can be instances of this generalised `Functor` class.

```
instance Functor [] where
  constraint Inv [] e = ()
  fmap = ...
```

```
instance Functor Set where
  constraint Inv Set e = Ord e
  fmap = ...
```

A *default constraint* can be given in a class declaration for an associated constraint family, similar to default method implementations. Any instance that does not explicitly provide an associated constraint opts for the default constraint. For `Functor` we can use the `()` constraint:[5]

[5] Recall that `() =>` τ is equivalent to the unqualified type τ.

```
class Functor f where
  constraint Inv f e = ()
```

This is particularly convenient because it allows reuse of the existing `Functor` instances without modification. In contrast, work-arounds such as Restricted Monads [11] require rewriting existing instances.

Final Polymorphic EDSL The final polymorphic EDSL becomes much more useful with constraint families:

```
class Expr sem where
  constraint Pre sem a
  constant :: Pre sem a => a -> sem a
  add      :: Pre sem a => sem a -> sem a -> sem a

data E a = E {eval :: a}

instance Expr E where
  constraint Pre E a = Num a
  constant c = E c
  add e1 e2  = E $ eval e1 + eval e2
```

Semantics for the EDSL, provided by the `sem` type, are free to choose their own constraints with an instance of the `Pre` constraint family, thus opening up a much wider range of applications.

HOAS-based EDSL Due to type synonym families, Haskell programs no longer have principal types. As a remedy, explicit equality constraints $\tau_1 \sim \tau_2$ were added to recover principal typing. For instance, consider:

```
class Coll c where
   type Elem c
   insert :: c -> Elem c -> c

addx c = insert c 'x'
```

The principal type of `addx` is `(Coll c, Elem c ~ Char) => c -> c`, which cannot be expressed without the explicit equality constraint. We may wonder whether, similarly, an explicit implication constraint, say C `|=` D, between constraints C and D is necessary for principality or expressivity reasons. For instance, in order to generalise the HOAS-based EDSL, we may want to write:

```
constraint family TermF s t

data Tree s where
  NewVar :: (TermF s t, TermF s t |= Term s t) => (t -> Tree s) -> Tree s
```

which expresses 1) that the constraint relating `s` and `t` is indexed by the solver `s` and term type `t`, and 2) that the constraint implies `Term s t`. Hence, this code is strictly more general than the previous version in Section 2.2. Yet, the only way to use (or *eliminate*) such an explicit implication is through a *modus ponens* rule:

$$\texttt{(TermF s t, TermF s t |= Term s t)} \;\models\; \texttt{Term s t}$$

This suggests a simpler solution that does not involve an explicit implication `|=`, but directly requires the right-hand side of the explicit implication:

```
constraint family TermF s t

data Tree s where
  NewVar  :: (TermF s t, Term s t) => (t -> Tree s) -> Tree s
```

This solution expresses the indexing property 1) and that `Term s t` holds, without enforcing a relationship between `TermF s t` and `Term s t`.

5.2 Well-defined Families

For constraint families to be well-defined, we must consider *confluence* and *termination*. The former ensures that family reductions are unambiguous, the latter ensures that they are well-founded.

Confluence A constraint family application which reduces to two distinct constraints is ambiguous and may ultimately lead to program crashes. For instance, in a dictionary-based implementation, ambiguity of constraints results in ambiguity of dictionary types, which may cause crashes at runtime if we lookup a non-existant method or method of an unexpected type. Hence, we enforce confluence by requiring non-overlapping instances, in the same way as type families [2]. This means that at most one instance matches a family application. Consequently, type family applications are not allowed as parameters of constraint family instances; due to their *openness*, overlap cannot be ruled out.

Termination The termination of synonym family reductions is more complicated than that of synonyms because family definitions are *open*. Even if a family's call graph is acyclic, further modules may define cycle-forming instances, possibly causing infinite reductions. However, not all cyclic call graphs are nonterminating; for instance, the following constraint family is terminating:

```
constraint family K (m :: * -> *) a
constraint instance K []           a = ()
constraint instance K Set          a = Eq a
constraint instance K (StateT s m) a = K m a
```

Synonyms may hide family constructors, thus all (non-family instance) synonyms should be substituted prior to well-definedness checking of families.

Because termination checking is generally undecidable, GHC imposes conservative conditions on type synonym families, some of which are discussed in recent work [4]. These conservative conditions can be applied to constraint families to ensure termination, which we present below as the *strong termination condition*. However, due to the nature of constraints, it is possible to relax these constraints allowing greater expressivity whilst still ensuring termination. We first define the strong termination condition and, motivated by examples, go on to successively weaken the condition.

Definition 1 (Strong Termination Condition). *For each constraint family instance $K\ \bar{\tau} = C$,*

1. *either C contains no constraint family application, or*
2. *C is a constraint family application of the form $K'\ \bar{\tau}'$, and*
   (a) $|\bar{\tau}| > |\bar{\tau}'|$,
   (b) the RHS has no more occurrences of any type variables than the LHS,
   (c) the RHS does not contain any type family applications

   The size measure $|\cdot|$ is defined as:

$$|a| = 1 \qquad |(\tau_1\ \tau_2)| = |\tau_1| + |\tau_2|$$
$$|T| = 1 \qquad |\bar{\tau}| = \textstyle\sum_{\tau \in \bar{\tau}} |\tau|$$

The above example satisfies the strong termination condition and is rightly accepted as terminating. The following terminating instance is also accepted as it contains no constraint family applications (satisfying case *1*), although it does apply a type family in an equality constraint:

```
constraint instance K (State s) = (Eq [s], s ~ F (s,s))
type family F s
```

The following non-terminating instances are on the other hand rejected:

```
constraint instance K Foo a = K Bar a
constraint instance K Bar a = K Foo a
```
} violates *2 (a)*

```
constraint instance K (Baz [x] m) a = K (Baz a m) a
```
} violates *2 (b)*

```
constraint instance K (Foz (Foz m)) a = K (F m) a
type family F (m :: * -> *)
type instance F Set = Foz (Foz Set)
```
} violates *2 (c)*

where all of `Foo`, `Bar`, `Baz` and `Foz` are data type constructors. In the second case, non-termination occurs when `a` is of the form `[x]`. In the third case, `K (Foz (Foz Set)) a` is non-terminating; the type family `F` reduces such that the left-hand side and right-hand side of the constraint instance are equal, hence forming a non-terminating family reduction.

The strong termination condition is however too conservative for our purposes, disallowing many common, terminating constraint families. For example, the following contains more than one family occurrence in the right-hand side:

```
constraint family K a
constraint instance K (a,b)  = (K a, K b)
```

yet this instance is terminating. Contrast this with a type family instance of a similar form:

```
type family TF a
type instance TF (a,b)  = (TF a, TF b)
```

where the constraint α `~ TF (`α`,Int)` would lead to an infinite type checker derivation. The problem is that the only solution for the unknown type α is an infinite type term `(((..., TF Int), TF Int), TF Int)`, which is built up gradually by instantiating α. Such a problem does not arise in the constraint family setting for two reasons: 1) an unknown *type* α is never bound to a *constraint* constructor `(,)`, and 2) there are no equality constraints between constraints.

Thus, we can impose a less severe termination condition for constraint families than for type families.

Definition 2 (Weak Termination Condition). *For each constraint family instance $K\ \bar{\tau} = C$, for each constraint synonym family application $K'\ \bar{\tau}'$ in C:*

1. $|\bar{\tau}| > |\bar{\tau}'|$,
2. *τ' has no more occurrences of any type variables than the left-hand side,*
3. *τ' does not contain any type family applications.*

Finally, we consider the interaction of constraint families with class instances, which also forms a derivation system. Allowing constraint families in contexts of class instances permits non-termination via mutual recursion. Consider:

```
class K a                          constraint family KF a
instance KF a => K [a]             constraint instance KF a = K [a]
```

which exhibits looping derivation `K [b]` $\rightarrow$ `KF b` $\rightarrow$ `K [b]` $\rightarrow \ldots$. We see two ways to avoid this form of non-termination. The first solution is to disallow constraint synonym families altogether in instance contexts; i.e. ruling out the above type class instance. The second solution is to strengthen the termination condition for family instances; i.e. ruling out the above family instance.

Definition 3 (Class Instance Compatible Termination Condition). *In addition to satisfying the Weak Termination Condition, we have for each constraint family instance $K\ \bar{\tau} = C$, that for each type class application $K'\ \bar{\tau}'$ in C the following three conditions are met:*

1. $|\bar{\tau}| \geq |\bar{\tau}'|$,
2. *τ' has no more occurrences of any type variables than the left-hand side,*
3. *τ' does not contain any type family applications.*

Note that the first condition requires a non-strict size decrease, rather than a strict one. The reason is that the Paterson conditions for termination of type class instances [12, Def. 11], already require a strict decrease from instance head to the individual constraints in the instance context. As a consequence, a strict decrease is still realised for mutual recursion between family and class instances.

6 Encoding and Implementation

As with type classes, both constraint synonyms and constraint families do not extend the power of the language but can be encoded using existing language

features. Such encodings are often much less convenient to write by hand than our proposed extensions. In this section we present an encoding which is employed by a prototype preprocessor accompanying this paper[6]. Appendix C illustrates a more direct implementation in System F_C. An alternate encoding is provided in Appendix D.

6.1 Constraint Synonym Encoding

Encoding constraint synonyms using existing features is considerably lightweight. A constraint synonym can be encoded as a class declaration with a single catch-all instance e.g.

```
class (Additive a, Multiplicative a, FromInteger a) => Num a where ...
instance (Additive a, Multiplicative a, FromInteger a) => Num a where ...
```

However, this approach requires GHC's *undecidable instances* extension, removing conservative termination conditions on type class instances. This extension is globally applied, thus type-checking decidability can no longer be guaranteed – an unnecessary, undesirable side-effect of this encoding. An alternative to using *undecidable instances* is to supply individual instances for each required type. This is tedious and even more inelegant than the above auxiliary class definition.

6.2 Constraint Synonym Family Encoding

The encoding of constraint synonym families relies on two ingredients: 1) type synonym families (below `Pre`) for capturing the synonym family aspect, and 2) GADTs (below `NumDict`) reifying constraints in a type-level construct. Applied to the final polymorphic EDSL we obtain the following solution:

```
type family Pre (sem :: * -> *)  a

class Expr sem where
  constant :: Pre sem a -> a -> sem a
  add      :: Pre sem a -> sem a -> sem a -> sem a

data E a = E {eval :: a}
type instance Pre E a = NumDict a
data NumDict a where ND :: Num a => NumDict a

instance Expr E where
  constant _ c  = E c
  add ND e1 e2  = E $ eval e1 + eval e2
```

This encoding is much more cluttered than the original due to the value-level passing of reified constraints and the releasing of constraints by GADT pattern matching.

Polymorphic expressions are equally cluttered. For instance,

[6] `http://github.com/dorchard/constraintTermExtensions`

```
exp :: (Expr sem, Pre sem Int) => sem Int
exp d = add d (constant d 1) (constant d 2)

three :: Int
three = eval exp
```

becomes:

```
exp :: Expr sem => Pre sem Int -> sem Int
exp d = add d (constant d 1) (constant d 2)

three :: Int
three = eval (exp ND)
```

7 Related Work

Proposed *class aliases* (or *context aliases*) [6, 7] can define constraint synonyms, but have extra features to ease refactoring. Class aliases define a new class from a conjunction of existing classes, whose methods are at least those of the aliased classes. Additional methods may also be defined in the class alias. Class alias instances implement all methods of the class alias, e.g.

```
class alias Num a = (Additive a, Multiplicative a, FromInteger a) where
    (-) :: a -> a -> a
instance Num Integer where
     x + y = ...
     x - y = ...
```

Existing instances of `Num` do not have to be rewritten as individual instances of `Additive`, `Multiplicative`, unlike an equivalent constraint synonym.

However, some class aliases are potentially problematic:

```
class alias Eq' a b = (Eq a, Eq b)
```

Instances of `Eq'` must implement two equality operations, although the type to which each belongs may be indistinguishable for some instances.

Another issue arises if class instances overlap class alias instances e.g. if both `Additive Int` and `Num Int` are defined, which implementation is chosen?

Constraint synonyms are more general than class aliases, simply extending type synonyms to the constraint-level. A class alias-like extension to constraint synonyms might allow class instances of constraint synonyms.

Restricted data types [13] were proposed to address the problem of writing polymorphic classes whose instance parameters may have constraints. Restricted data types allow constraints on the parameters of a data type to be given with a data type definition; constraints on a type are implicit. Such an approach is less flexible than our own, not permitting arbitrary type-indexing of constraints, or more specialised constraints under certain type parameters.

An encoding for restricted monads [11] binds all method parameters in the head of a class declaration such that instance-specific constraints can be given.

Such an approach is not as practical as constraint families, as it requires all existing instances to be rewritten, and cumbersome class declarations, with potentially many subclasses to differentiate methods, and many parameters.

The RMonad library [14] gives restricted alternatives to *functor* and *monad* classes on which instance-specific constraints can be given, using a similar manual encoding to Section 6.2.

Class families and *constraint families* have been discussed informally in blog posts and online discussions [8, 15]. We feel that class families provides unnecessary machinery to solve a problem that constraint synonym families solve more elegantly (see Appendix A). In online discussions, design issues (such as default values, extra non-class parameters of associated families, redundancy of explicit implications) and semantical aspects (static semantics, termination) have not been considered thoroughly until now.

Appendix B gives an object-oriented (Java) encoding of the polymorphic EDSL example as a comparison to Haskell's type system features.

8 Conclusion & Further Work

This paper highlights the current imbalance in the Haskell/GHC type system, at the disadvantage of type-level constraints. This imbalance imposes a rather unfortunate barrier for building larger and more flexible systems, including polymorphic EDSLs, in Haskell. The balance is restored by transferring the term constructs of synonyms and families to the constraint language. This symmetrising approach provided a reference syntax and semantics from which to derive the new constraint-level features, such that their design was not "from scratch".

We used a three-dimensional design space to characterise existing type system features and to elucidate the imbalance between types and constraints. It would be interesting to see if such a framework could not only characterise features, but provide a systematic approach to defining syntax, semantics, properties, and even implementations, of type system features. Type system features could be defined in terms of of disjoint, abstract units. There are certainly further interesting axes of such a design space, such as *open* vs. *closed* definitions.

As programming pervades science, engineering, and business, and as new (parallel) hardware architectures emerge, the utility of DSLs is becoming increasingly apparent. Building DSLs within a well-established language provides inexpensive application or implementation specific expressivity and optimisation. A good EDSL-host language must be proficient at handling abstract structures, with high levels of parametricity. Our extensions increase Haskell's ability to host EDSLs, further boosting its potential.

Acknowledgements We are grateful to Manuel Chakravarty and Simon Peyton Jones for discussions and explaining details of GHC's associated type families. Also thanks to Max Bolingbroke, Oleg Kiselyov, Martin Sulzmann and Marko van Dooren for their insightful feedback.

References

1. Peyton Jones, S., et al.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (May 2003)
2. Chakravarty, M.M.T., Keller, G., Jones, S.P.: Associated type synonyms. In: ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2005) 241–253
3. Chakravarty, M.M.T., Keller, G., Jones, S.P., Marlow, S.: Associated types with class. SIGPLAN Not. **40**(1) (2005) 1–13
4. Schrijvers, T., Jones, S.P., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. SIGPLAN Not. **43**(9) (2008) 51–62
5. Stewart, D.: Domain Specific Languages for Domain Specific Problems. In: Workshop on Non-Traditional Programming Models for High-Performance Computing, LACSS. (2009)
6. Meacham, J.: Class Alias Proposal for Haskell (last visited August 2009) `http://repetae.net/recent/out/classalias.html`.
7. Jeltsch, W., van Dijk, B., van Dijk, R.: HaskellWiki: Context alias entry (last visited August 2009) `http://www.haskell.org/haskellwiki/Context_alias`.
8. Chakravarty, M., Peyton Jones, S., Sulzmann, M., Schrijvers, T.: GHC developer wiki: Class families entry (last visited August 2009) `http://hackage.haskell.org/trac/ghc/wiki/TypeFunctions/ClassFamilies`.
9. Schrijvers, T., Stuckey, P., Wadler, P.: Monadic Constraint Programming. J. Func. Prog. **19**(6) (2009) 663–697
10. Carette, J., Kiselyov, O., Shan, C.: Finally Tagless, Partially Evaluated. In Shao, Z., ed.: APLAS. Volume 4807 of LNCS., Springer (2007) 222–238
11. Kiselyov, O.: Restricted Data Types Now (February 2006) `http://okmij.org/ftp/Haskell/RestrictedMonad.lhs`.
12. Sulzmann, M., Duck, G.J., Peyton-Jones, S., Stuckey, P.J.: Understanding functional dependencies via constraint handling rules. J. Func. Prog. **17**(1) (2007) 83–129
13. Hughes, J.: Restricted Data Types in Haskell. In: Proceedings of the 1999 Haskell Workshop. Technical Report UU-CS-1999-28, Utrecht (1999)
14. Sittampalam, G., Gavin, P.: rmonad: Restricted monad library (2008) `http://hackage.haskell.org/package/rmonad`.
15. Bolingbroke, M.: Constraint families (2009) `http://blog.omega-prime.co.uk/?p=61`.
16. Kiselyov, O.: Haskell with only one typeclass (February 2007) `http://okmij.org/ftp/Haskell/types.html#Haskell1`.

A Class Families

Class families are the missing type feature in our extrapolation of type-level features to the constraint-level (see **Fig.1**), mirroring families of generative type definitions. There has been some online discussion of class families [8] which fit this gap in the design space. Class families have been proposed as a solution to the `Functor` problem that we solved adequately with constraint synonym families.

Given class families, an associated class family may solve the `Functor` problem as such:

```
class Functor f where
   class Inv f e

instance Functor Set where
   class Ord e => Inv Set e

instance Ord e => Inv Set e
```

The superclass constraint `Ord e` emulates the effect of a constraint synonym family. Class family instance methods are redundant in this approach. Moreover, a tiresome instance is required for each class in the family. Hence, all in all, the constraint synonym family is a much more elegant solution.

In contrast to data type families versus type synonym families, we do not see any appealing use of class families over constraint synonym families. The main difference bewteen data type/class families and type/constraint synonym families is that the former are *injective*. Injectivity for a type/constraint constructor T means, that if $T\ \alpha$ and $T\ \beta$ are equivalent, then so are α and β. Injectivity for type terms can be exploited because type equivalence constraints $T\ \alpha \sim T\ \beta$ actually arise. In contrast, equivalence between constraints does not occur during type inference, only implication. From $C\ \alpha \Rightarrow C\ \beta$ we cannot conclude that $\alpha \sim \beta$. Hence, the injectivity of class families does not pose an advantage over constraint synonym families.

B The Polymorphic Final EDSL in OOP

OOP languages like Java succeed better at unifying type term and type constraints. For instance, the expression EDSL example in this paper does not require novel language features. Here is a possible implementation:

```
abstract class Exp<A, Sem extends Exp<A,Sem>> {
  abstract Sem constant(A a);
  abstract Sem add(Sem e1, Sem e2);
}

class E<A extends Num<A>> extends Exp<A, E<A>> {
  A eval;
  E(A a) { eval = a; }
```

```
  E<A> constant(A a) { return new E<A>(a); }
  E<A> add(E<A> e1, E<A> e2) {
   return new E<A>(e1.eval.plus(e2.eval));
  }
}

interface Num<A extends Num<A>> {
  A plus(A other);
}
```

This implementation maps both Haskell types, like `E`, and Haskell type classes, `Exp` and `Num`, to Java types. Thus, type class instantiation becomes subclassing/interface implementation. Type class constraints imposed on type variables are expressed as subtyping bounds, e.g. `<A extends Num<A>>` expresses Haskell's `forall a. Num a => ...`.

The self F-bounded polymorphism on `Exp` and `Num` is necessary to deal with non-unary functions. For instance, without the F-bound on `Sem`, we would not know in `E` that the parameters to `add` also have type `E`, and consequently have the method `eval`.

C System F_C Encoding

In this section we illustrate a direct System F_C encoding of associated constraint families on the final polymorphic EDSL example seen throughout this paper.

A type class declaration is encoded as a *dictionary*, a data type with one constructor whose fields are the type class methods.

```
data Expr sem = Expr (∀a. Pre sem a -> a -> sem a)
                     (∀a. Pre sem a -> sem a -> sem a -> sem a)
```

Type class methods are encoded as field selectors of the dictionary:

```
constant = Λsem.λ(d:Expr sem).case d of { Expr m1 m2 -> m1 }
add      = Λsem.λ(d:Expr sem).case d of { Expr m1 m2 -> m2 }
```

The `E` data type and its field selector are encoded in the standard way:

```
data E a = E a

eval     = Λa.λ(e : E a).case e of { E v -> v }
```

Finally, the `Exp E` instance is encoded as a dictionary constructing function.

```
exprE :: Expr E
exprE = Expr (Λa.λ(d:Pre E a).λ(c:a).E c)
             (Λa.λ(d:Pre E a).λ(e1:E a).λ(e2:E a).
                 E ((+) @a (d ▶ PreE a) (eval @a e1) (eval @a e2)))
```

Note that each method field takes a dictionary `d` of type `Pre E a`. The associated constraint synonym expresses that it equals `Num a`. This is captured in a new equality axiom between the corresponding dictionary types.

```
axiom CoPreE :: ∀a. Pre E a ~ Num a
```

Note how this axiom is used in the encoding of the `add` method. In order to call `(+)` at type `a`, the encoding must supply a dictionary of type `Num a`. Such a dictionary is obtained by casting `d` of type `Pre E a` using the coercion `CoPreE a`.

D Alternate Constraint Synonym Families Encoding

In this section we illustrate an alternate encoding of constraint synonym families, provided to us by Oleg Kiselyov, using a single-typeclass approach to defining all Haskell classes [16]. The following example[7] encodes the final polymorphic EDSL example seen throughout this paper.

A single class, `Apply`, is used to define all other classes:

```
class Apply label r | label -> r where
    apply:: label -> r
```

where `label` is a type labelling a method of the encoded class, and `r` is the type of that method. The `apply` function returns the method implementation given a dummy value of the `label` type.

Because the `Expr` class has two methods, we require two label types:

```
data Const (sem :: * -> *) a = Const
data Add   (sem :: * -> *) a = Add
```

The methods `constant` and `add` give the signature of the unencoded method in their type term; in the constraint terms the function is labelled and, by the `Apply` class, related to the unencoded method's signature.

```
constant :: forall sem a. Apply (Const sem a) (a -> sem a) => (a -> sem a)
constant = apply (Const :: Const sem a)

add :: forall sem a. Apply (Add sem a) (sem a -> sem a -> sem a) =>
       sem a -> sem a -> sem a
add = apply (Add :: Add sem a)
```

Each instance of `Expr` is encoded as two instances of `Apply`, one for each unencoded method's corresponding label and signature types. Parameters of label and signature types are bound to the unencoded instance's type parameters. The implementation of `apply` gives the operation of the unencoded method.

```
data Eval a = E { eval :: a }

instance Apply (Const Eval a) (a -> Eval a) where
```

[7] `http://okmij.org/ftp/tagless-final/PolymorphicEDSL.hs`

```
    apply _ = E

instance Num a => Apply (Add Eval a) (Eval a -> Eval a -> Eval a) where
    apply _ = \e1 e2 -> E $ eval e1 + eval e2
```

It is in these instances that there is the flexibility to add instance-specific constraints (in this case `Num a`) to the methods. Another set of instances gives a different semantics to the EDSL, where terms are strings.

```
data S a = S { sh :: String }

instance Show a => Apply (Const S a) (a -> S a) where
    apply _ = S . show

instance Apply (Add S a) (S a -> S a -> S a) where
    apply _ = \e1 e2 -> S $ sh e1 ++ sh e2
```