Chapter 4

One Vote for Type Families in Haskell!

Louis-Julien Guillemette*, Stefan Monnier*
Category: Position

Abstract: Generalized Algebraic DataTypes (GADTs) allow programmers to capture important invariants of their data structures through type annotations on data constructors. However when working with GADTs, it is often difficult to concisely and precisely express the way complex data manipulations maintain those invariants. One can approach the problem in a few different ways, given the arsenal of GHC's current type extensions. One approach is to use type classes with functional dependencies. Another approach is to introduce yet more GADTs to capture the behavior of individual functions. A third alternative is to use type families, a recent introduction of GHC by which functions over types can be defined directly, much like term-level functions, and appear in type signatures.

In this paper we illustrate the use of type families in the context of a type-preserving compiler written in Haskell. We compare the results with an *ad-hoc*, all-GADT solution. We argue that type families promote a more direct programming style that eliminates much code bloat and translates into increased run-time performance. They offer better modularity and require fewer type annotations than type classes, which require that class constraints be propagated between compilation phases.

We also describe a use of type families to capture more complex data structure invariants. We mention current limitations that we face with these more advanced uses, in which we need to convince the type checker that the type families we define satisfy certain properties. We sketch a proposal of a language extension to directly support such properties.

*Université de Montréal, C.P. 6128, succ. Centre-Ville, Montréal, Québec, Canada.
++1 (514) 343-6111 x3545; {guillelj,monnier}@iro.umontreal.ca

4.1 INTRODUCTION

There is a definite trend in richly typed functional languages to incorporate features from dependently typed languages and proof assistants. In the world of Haskell, an important step in this direction has been the introduction of Generalized Algebraic Datatypes (GADTs) in GHC. GADTs can be seen as a restricted form of dependent types, where the stage-distinction between terms and types is not lost. They allow data constructors to bear extra type annotations that can describe the shape, size, content, or other properties of the data. These annotations can be used to express key invariants of the data structures, and thus rule out invalid uses of the data.

Even before GADTs were around, the rich type system of Haskell and its proposed extensions lent itself to simulating dependent typing. Typically this could be done with sophisticated uses of type classes (as in [4]), or even plain old parametric polymorphism (as in [13]). But as GADTs extend the basic notion of data declaration at the core of the language, they promise to support a sort of dependent programming in a more direct manner. But while GADTs constrain the data that can be constructed, they do not provide a particular way of capturing in the types the way such invariants are satisfied as new data is created from old one. Multi-parameter type classes can sometimes serve that purpose, although the results are often contrived and difficult to understand. Alternatively, as GADTs essentially encode relations on types, they can be used for relating the types of the inputs and outputs of a function. This approach is particularly flexible, but tends to be rather verbose and incurs run-time overhead.

There has recently been a proposal to extend Haskell with functions over types, or so-called *type families* [17]. They allow the programmer to define functions over types by case analysis, and refer to these functions in the signatures of functions (or data constructors.) This development has been introduced in GHC and we could thus experiment with those new tools. Our experience has generally been that precise relationships between types can be expressed succinctly and precisely, while achieving the same effect using other techniques requires rather more indirect and elaborate encodings.

Type families are a generalization of associated types [3], in the sense that they can be defined independently of type classes. The question whether associated types or the alternative approach of multi-parameter type classes with functional dependencies [11], should make it to the next Haskell standard [16], is one of the most hotly debated issues in the standardization effort. This motivated us to document our experience and take a position in favor of type families.

4.1.1 Context

We have been using GADTs extensively in the context of a type-preserving compiler for a functional language [7, 8]. The primary use we make of GADTs is to encode program representations of our source and intermediate languages, in a way that enforces each language's type system. As a starting point, the simply

```
data ValK t where
  LamK :: (ValK t → ExpK) → ValK (t → Void)
  PairK :: ValK s → ValK t → ValK (s,t)

data ExpK where
  AppK :: ValK (t → Void) → ValK t → ExpK

data Void
```

FIGURE 4.1. Encoding of the CPS language

typed λ-calculus can be encoded as:

```
data Exp t where
  Lam :: (Exp s → Exp t) → Exp (s → t)
  App :: Exp (s → t) → Exp s → Exp t
```

In contrast to an ordinary algebraic datatype, a GADT definition can have data constructors of varying type. Here, the type idExp has a type parameter t that reflects the source type of the expression: a Haskell term of type *Exp t* encodes an expression of source type τ, where t is the Haskell type we use to reflect τ. We can also view *Exp* as an encoding of type derivations rather than just expressions, as type derivations are in one-to-one correspondence with well-typed expressions.

The tricky part is to give a type to a function that implements a transformation over idExp. We will take the example of the CPS conversion, as its theory is well understood (see e.g. [2, 15]). At first approximation, its type would be:

```
cps :: Exp t → (ValK t' → ExpK) → ExpK
```

where *ValK t* is a value in CPS of type t and *ExpK* is a well-formed CPS expression; the two datatypes are defined in Fig. 4.1. The second argument to *cps* is a continuation: it is a function that expects a value of object type t', consistent with the source type t. The relationship between t and t' is captured by a function mapping source types to types in CPS:

$$\mathscr{K}[\![\tau_1 \to \tau_2]\!] = (\mathscr{K}[\![\tau_1]\!], \mathscr{K}[\![\tau_2]\!] \to \textbf{void}) \to \textbf{void}$$

In the sections that follow, we will see different implementations of *cps* using different types. We will focus on the rule that CPS-converts a function application[1]:

$$\mathscr{K}_{\text{exp}}[\![e_1\ e_2]\!]\ k = \mathscr{K}_{\text{exp}}[\![e_1]\!]\ (\lambda v_1.\ \mathscr{K}_{\text{exp}}[\![e_2]\!]\ (\lambda v_2.\ v_1\ \langle v_2, k\rangle))$$

Outline The rest of this paper is structured as follows. We first introduce type families and implement the function *cps* using these (Sec. 4.2). We then consider

[1] Restricting to the case of function application rather than function abstraction relieves us from the intricacies of dealing with binders, so as to better focus on type issues.

an alternative solution that uses only GADTs (Sec. 4.3), and one that uses type classes (Sec. 4.4). We describe a more advanced use of type families for encoding more sophisticated typed languages and the some limitations thereof (Sec. 4.5), and sketch a proposal to address these limitations (Sec. 4.6).

4.2 TYPE FAMILIES

Type families allow us to directly define functions over types by case analysis, in a way that resembles term-level function definitions with pattern matching. For example, we can define a type function *Add* that computes (statically) the sum of two Peano numbers:

```
data Z; data S i          — natural numbers encoded as types

type family Add n m
type instance Add Z m = m
type instance Add (S n) m = S (Add n m)
```

We can then use this type family to express the fact that an *append* function over length-annotated lists produces a list of the expected length:

```
data List elem len where
  Cons :: elem → List elem n → List elem (S n)
  Nil :: List elem Z

append :: List elem n → List elem m → List elem (Add n m)
append Nil l = l
append (Cons h t) l = Cons h (append t l)
```

Note that the definition of *append* follows the structure of the definition of the type function *Add*, so that the type checker can verify that every clause of *append* satisfies its type signature. We can define $\mathscr{K}[\![-]\!]$ similarly:

```
type family CPS t
type instance CPS (s → t) = ((CPS s, CPS t → Void) → Void
```

and we can refer to it in the type of cps:

```
cps :: Exp t → (ValK (CPS t) → ExpK) → ExpK
```

We can then implement cps in the most straightforward style and have the type-checker verify that the constraints on object types are respected:

```
cps (App e1 e2) k =
  cps e1 (λv1 →
    cps e2 (λv2 →
      AppK v1 (PairK v2 (LamK k))))
```

In this example we get GHC's type checker to verify that our CPS conversion is type preserving, *for free*: were we to use a plain algebraic datatype instead of a GADT and not bother to enforce type preservation, the code of cps would be identical.

```
data CPS t t' where
  CpsFun :: CPS s s' → CPS t t'
            → CPS (s → t) ((s', t' → Void) → Void)

data CPSterm t where
  CPSterm :: CPS t t' → ((ValK t' → ExpK) → ExpK) → CPSterm t

cps :: Exp t → CPSterm t
cps (App e1 e2) =
 case cps e1 of
   CPSterm (CpsFun ss' tt') e1' →
     case cps e2 of
       CPSterm ss2' e2' →
         case cpsUnique ss' ss2' of
           EqRefl → CPSterm tt' (λk → e1 (λv1 → e2 (λv2 →
                                  AppK v1 (PairK v2 (LamK k)))))

data Equal a b where
  EqRefl :: Equal a a

cpsUnique :: CPS t t' → CPS t t'' → Equal t' t''
cpsUnique = ...
```

FIGURE 4.2. All-GADT CPS conversion

4.3 MORE GADTS

In the days when type families were not available to us, a workable solution (which we did use extensively) was to encode the type function $\mathscr{K}[\![-]\!]$ as relation using a GADT, and have *cps* produce an existential package containing a proof that the output term expected a continuation of suitable type. The type of *cps* then looks like:

$$cps :: Exp\ t \to (\exists t'.\ (CPS\ t\ t', (ValK\ t' \to ExpK) \to ExpK))$$

A term of type *CPS t t'* encodes a proof that $\mathscr{K}[\![\tau]\!] = \tau'$, where t encodes τ and t' encodes τ'. The above signature for *cps* actually abuses Haskell notation: in reality we need to introduce another GADT (*CPSterm t*) to bind an existential type and couple the term with the proof that it is of the expected type. The actual implementation is shown in Fig. 4.2.

As $\mathscr{K}[\![-]\!]$ is encoded as a relation, we need a separate proof that this relation is a function. This is accomplished by the *cpsUnique* function, which produces a witness that $\tau' = \tau''$, given proofs that $\mathscr{K}[\![\tau]\!] = \tau'$ and $\mathscr{K}[\![\tau]\!] = \tau''$. We call this function when converting a function application to ensure that the function and its argument are of compatible type.

In comparison to our initial solution with type families, the one shown here is unsatisfactory:

- It is cluttered with manipulations of existential packages. As a result, the code doing the translation roughly doubles in size.
- It requires a number of additional artifacts, such as the type *CPSterm* and the function *cpsUnique*.
- Of course constructing and inspecting the existential packages incurs run-time overhead.
- The "proofs" that the types match are encoded in an unsound logic, given Haskell's inherent ability to diverge.

4.4 TYPE CLASSES

Type classes are meant to support *ad-hoc* polymorphism: they allow the programmer to define functions that behave differently at different types. It is not immediately clear that this feature can be useful in our case: after all, cps proceeds by case analysis over the syntactic constructs, not the types. But there may be some indirect use of type classes by which we can express the way our syntax-directed translation produces terms of the expected type.

Multi-parameter type classes with functional dependencies allow us to define some form of intentional type functions. For example, we can express the type function $\mathscr{K}[\![-]\!]$ as follows:

$$\textbf{class } \mathit{CPS}\ t\ t' \mid t \to t'$$
$$\textbf{instance } (\mathit{CPS}\ s\ s', \mathit{CPS}\ t\ t') \Rightarrow \mathit{CPS}\ (s \to t)\ ((s', t' \to \mathit{Void}) \to \mathit{Void})$$

The first line declares a type class *CPS* as a relation between two parameters t and t', and states that t' is uniquely determined by t, so in effect we have a function from t to t'. The second line defines the function at $s \to t$. For type classes to be useful, they are normally equipped with member functions. We might try to define cps in this way:

$$\textbf{class } \mathit{CPS}\ t\ t' \mid t \to t'$$
$$\quad \textbf{where } cps :: \mathit{Exp}\ t \to (\mathit{ValK}\ t' \to \mathit{ExpK}) \to \mathit{ExpK}$$

Individual instance declarations are required to implement the member functions for the types they cover, for example:

$$\textbf{instance } (\mathit{CPS}\ s\ s', \mathit{CPS}\ t\ t') \Rightarrow \mathit{CPS}\ (s \to t)\ ((s', t' \to \mathit{Void}) \to \mathit{Void})$$
$$\quad \textbf{where } cps = \ldots$$

This example would give the translation for λ-abstractions. We would define other instances and implement cps for other introduction forms if we had any. But we cannot do the same for elimination forms (such as function application), as they are not identified with source types of a particular form.

One thing we can do is define the function cps separately from the class *CPS* and have it handle all the syntactic forms:

$$
\begin{aligned}
&cps :: CPS\ t\ t' \Rightarrow Exp\ t \rightarrow (ValK\ t' \rightarrow ExpK) \rightarrow ExpK \\
&cps\ (App\ e_1\ e_2)\ k = \\
&\quad cps\ e_1\ (\lambda v_1 \rightarrow \\
&\qquad cps\ e_2\ (\lambda v_2 \rightarrow \\
&\qquad\quad AppK\ v_1\ (PairK\ v_2\ (LamK\ k))))
\end{aligned}
$$

For this this to work, the compiler must know that there are instances of *CPS* that cover the types of e_1 and e_2. This forces us to add a class context to the constructor *App*:

$$App :: CPS\ s\ s' \Rightarrow Exp\ (s \rightarrow t) \rightarrow Exp\ s \rightarrow Exp\ t$$

Note that we do not need to mention $CPS\ t\ t'$ in this context, because the return type of *App* is $Exp\ t$, and we already have $CPS\ t\ t'$ in the context of cps.

The problem with these class constraints is that they embed knowledge about the CPS translation into the source language. So earlier phases get polluted by constraints that are specific to the later CPS phases. In general these class constraints propagate from the CPS conversion phase all the way to the type checking or type inference phase, which is the only phase where the types are sufficiently ground to make it possible to create the corresponding proofs. Of course, other compilation phases such as closure conversion or hoisting would require similar class constraints on their data constructors, which would propagate to the front-end as well. Class inheritance might sometimes be used to combine or synthesize these. But there is an inherent lack of modularity in this scheme, which amounts to pre-computing in some previous phase the type-level translation of a subsequent phase.

4.5 FURTHER USES OF TYPE FAMILIES

Type families are also useful to capture more complex invariants of data structures. We constructed an encoding of System F, where a set of type families implement substitution over System F types. In this section we describe this encoding as well as the difficulties we encountered when implementing code transformations over it.

Our representation of System F types encodes type variables (bound by $\forall$) as de Bruijn indices. For instance, the type of the flip function for pairs:

$$\forall \alpha, \beta.\ \langle \alpha, \beta \rangle \rightarrow \langle \beta, \alpha \rangle$$

would be represented as:

$$All\ (All\ ((Var\ (S\ Z), Var\ Z) \rightarrow (Var\ Z, Var\ (S\ Z))))$$

The data constructor for the intermediate language's representation of type application ($e[\tau]$) is defined as:

$$TpApp :: Exp\ (All\ s) \rightarrow Exp\ (Subst\ s\ t\ Z)$$

$$
\begin{array}{rcl}
(\forall\tau_0)[\tau/i] &=& \forall(\tau_0[\tau/i+1]) \\
j[\tau/i] &=& \begin{cases} j-1 & \text{if } j>i \\ U_0^i(\tau) & \text{if } j=i \\ j & \text{if } j<i \end{cases} \\
(\tau_1\to\tau_2)[\tau/i] &=& \tau_1[\tau/i]\to\tau_2[\tau/i] \\
\mathbf{int}[\tau/i] &=& \mathbf{int}
\end{array}
\qquad
\begin{array}{rcl}
U_k^i(\forall\tau) &=& \forall(U_{k+1}^i(\tau)) \\
U_k^i(j) &=& \begin{cases} j+i & \text{if } j>k \\ j & \text{if } j\le k \end{cases} \\
U_k^i(\tau_1\to\tau_2) &=& U_k^i(\tau_1)\to U_k^i(\tau_2) \\
U_k^i(\mathbf{int}) &=& \mathbf{int}
\end{array}
$$

FIGURE 4.3. Substitution over System *F* types

where *Subst* is the type family that implements substitution over types, defined below. This type encodes the usual typing rule for type application:

$$\frac{\Gamma\vdash e:\forall\alpha.\ \tau_1}{\Gamma\vdash e[\tau_2]:\tau_1[\tau_2/\alpha]}$$

With de Bruijn indices, we omit type variables in universal types, and substitution eliminates an index rather than a type variable, so the above rule would read:

$$\frac{\Gamma\vdash e:\forall\tau_1}{\Gamma\vdash e[\tau_2]:\tau_1[\tau_2/0]}$$

where 0 is the smallest de Bruijn index. The form $\tau[\tau'/i]$ yields the type τ where τ' has been substituted in place of the index i; substitution is formally defined in Fig. 4.3. It is a conventional substitution over de Bruijn terms (as in, e.g. [12]). It employs an "update" function $U_k^i(\tau)$ whose effect is to adjust all indices greater than k (those are the free variables) by incrementing them by i.

The substitution and update functions encode directly as Haskell type families. As their definition involve arithmetic over indices, we also need to define type functions accordingly. The complete list of type functions, with their meaning, is as follows:

type family *Subst* $t_1\ t_2\ i$	— $\tau_1[\tau_2/i]$	Substitute t_2 for i in t_1
type family $U\ k\ i\ t$	— $U_k^i(\tau)$	Add i to indices in t greater than k
type family *Pred* i	— $i-1$	Predecessor of i
type family *Add* $i\ j$	— $i+j$	Sum of i and j
type family *CMP* $i\ j\ t_1\ t_2\ t_3$	— $\begin{cases}\tau_1 & \text{if } i<j;\\ \tau_2 & \text{if } i=j;\\ \tau_3 & \text{if } i>j.\end{cases}$	

Henceforth, the definition of individual type families is straightforward:

```
type instance Subst All s t i = All (Subst s t (S i))
type instance Subst (Var j) t i = CMP i j (Var (Pred j)) (U Z i t) (Var j)
...
type instance U k i (All t) = All (U (S k) i t)
type instance U k i (Var j) = Var (CMP j k j j (Add j i))
...
```

4.5.1 Limitations

We were able to extend our CPS conversion (and subsequently the other phases as well) to work with this representation of polymorphism using type families. The technical difficulty it introduces is that some work is needed to convince the type checker that we obtain a well-typed term when converting a type application (or abstraction), as it involves reconstructing a term whose type is defined by a substitution. For instance, the translation of a type application is defined as:

$$\mathscr{K}_{\text{exp}}[\![e[\tau]]\!]\ k = \mathscr{K}_{\text{exp}}[\![e]\!]\ (\lambda x\ .\ x[\mathscr{K}[\![\tau]]\!]\ (\lambda y\ .\ k\ y))$$

The type safety of this rule relies on the fact that our notion of substitution commutes with the type translation:

Lemma 4.1. *(subst-$\mathscr{K}[\![-]\!]$ commute) For all source types τ_1, τ_2 and index i,*

$$\mathscr{K}[\![\tau_1[\tau_2/i]]\!] = \mathscr{K}[\![\tau_1]\!][\mathscr{K}[\![\tau_2]\!]/i].$$

To put this in context, it means that we need to make a coercion between the types:

$$\mathit{ValK}\ (\mathit{Subst}\ (\mathit{CPS}\ s)\ (\mathit{CPS}\ t)\ Z)$$

and

$$\mathit{ValK}\ (\mathit{CPS}\ (\mathit{Subst}\ s\ t\ Z))$$

for the supplied continuation (k) to be of a type compatible with the translated term. To provide this coercion, we have two options: we can add it to the context as a required type predicate, or implement the lemma as a term-level function.

Lemma in the context We can annotate the data constructor *TpApp* with a constraint stating that the lemma holds at the types in question.

$$\begin{aligned}&\mathit{TpApp} :: \mathit{CPS}\ (\mathit{Subst}\ s\ t\ Z) \sim \mathit{Subst}\ (\mathit{CPS}\ s)\ (\mathit{CPS}\ t)\ Z \Rightarrow\\ &\qquad \mathit{Exp}\ (\mathit{All}\ s) \to \mathit{Exp}\ (\mathit{Subst}\ s\ t\ Z)\end{aligned}$$

A constraint of the form $s \sim t$ means that the types s and t, although possibly syntactically different, are equivalent after applying a process of normalization (which in particular eliminates applications of type functions.) These *type equality coercions* [24] are another feature introduced in GHC along with type families.

We can then implement *cps* as follows[2]:

$$\mathit{cps}\ (\mathit{TpApp}\ e)\ k\ = \mathit{cps}\ e\ (\lambda x \to \mathit{TpAppK}\ x\ (\mathit{LamK}\ k))$$

This scheme basically moves the burden of proving the property to the point where the property is trivial to prove because s is known. This means it is propagated just like type class constraints in Sec. 4.4, and suffers from the same problems: the

[2]Note that type application is implicit in Haskell syntax, hence the difference from the definition of $\mathscr{K}_{\text{exp}}[\![-]\!]$.

proof that *Subst* and *CPS* commute ends up being constructed in the front end and propagated through the compiler pipeline until reaching the CPS phase. Also this has to be done for every such property we need, and it appears that we generally cannot combine or synthesize those proofs from each other using something like class inheritance, so we end up with very large type annotations throughout the compiler.

Lemma as a function An alternative solution is to implement the lemma as a term-level function, which produces a witness that the coercion is valid. Its type is:

$$\begin{aligned}&\mathit{substCpsCommute} ::\\ &\quad \mathit{TypeRep}\ s \to \mathit{TypeRep}\ t \to \mathit{NatRep}\ i\\ &\to \mathit{Equiv}\ (\mathit{CPS}\ (\mathit{Subst}\ s\ t\ i))\ (\mathit{Subst}\ (\mathit{CPS}\ s)\ (\mathit{CPS}\ t)\ i)\end{aligned}$$

$$\begin{aligned}&\textbf{data}\ \mathit{Equiv}\ s\ t\ \textbf{where}\\ &\quad \mathit{Equiv}\ ::\ s \sim t \Rightarrow \mathit{Equiv}\ s\ t\end{aligned}$$

The type *Equiv* reifies a type equality coercion at the term level, and generalizes the type *Equal* from Sec. 4.3. The lemma itself (*substCpsCommute*) can be defined by case analysis over runtime type representations (*TypeRep*) of the types s and t. Alternatively, it can do a dynamic test: it can construct a representation of the two types to prove equal and perform a comparison over them to supply evidence that they match.

Of course, in order to apply the lemma, we now need type annotations on the data constructor:

$$\mathit{TpApp} :: \mathit{TypeRep}\ s \to \mathit{TypeRep}\ t \to \mathit{Exp}\ (\mathit{All}\ s) \to \mathit{Exp}\ (\mathit{Subst}\ s\ t\ Z)$$

The implementation of cps is then:

$$\begin{aligned}&cps\ (\mathit{TpApp}\ sr\ tr\ e)\ k\ =\\ &\quad \textbf{case}\ \mathit{substCpsCommute}\ sr\ tr\ \textbf{of}\\ &\qquad \mathit{Equiv} \to cps\ e\ (\lambda x \to \mathit{TpAppK}\ x\ (\mathit{LamK}\ k))\end{aligned}$$

In addition to the type annotations on the syntax, implementing the coercions such as Lemma 4.1 at the term level has the downsides that the lemma itself is implemented in an unsound logic, and executing the lemma incurs run-time overhead.

4.5.2 The view from the other sides

Note that if we did not want to use type families, we could still encode type functions such as *CPS* and *Subst* as relations, using either GADTs or type classes, as we did in Sec. 4.3 and 4.4. For instance, the constructor for type application would look like:

$$\mathit{TpApp} :: \mathit{Subst}\ s\ t\ Z\ t' \to \mathit{Exp}\ (\mathit{All}\ s) \to \mathit{Exp}\ t'$$

or

$TpApp :: Subst\ s\ t\ Z\ t' \Rightarrow Exp\ (All\ s) \rightarrow Exp\ t'$

where $Subst\ s\ t\ i\ t'$ would be the relation such as $s[t/i] = t'$. This can make for potentially large type annotations, and suffers from the same inconveniences as discussed earlier. Especially, a GADT-based encoding would require proofs that the five relations defined are indeed functions.

In such a situation, the lemma we would need to prove would look like the following:

$$CPS\ s\ cps_s \wedge CPS\ t\ cps_t \wedge Subst\ cps_s\ cps_t\ Z\ cpssubst$$
$$\Leftrightarrow Subst\ s\ t\ Z\ subst \wedge CPS\ subst\ cpssubst$$

If the relations are encoded as GADTs, this lemma can be proved straightforwardly by writing the corresponding Haskell function. But it would of course incur a runtime cost, would need one function for each direction, and would still suffer from the fact that those proofs are written in an inconsistant logic.

If the relations are encoded as type classes, we cannot directly prove the lemma, and we probably cannot move the constraints into the context either, because we need these predicates to be valid for all t, a second-order quantification. The lemmas proposed in the next section for type families could probably be extended to apply to type classes as well, in which case they could probably be used here as well.

4.6 LEMMAS OVER TYPE FAMILIES

The previous section motivates the need for a facility by which some form of reasoning about type families could be carried out at the type level, so as to avoid the pitfalls of run-time checks without having type annotations encumber the whole compiler.

One difficulty with type families is that they are by definition *open*, i.e. nothing prevents a type family from being extended with instances for unforeseen types. This problem was already recognized in [18] where they point out that they cannot rely on properties such as $Sum\ n\ Zero = n$ since a new instance of *Sum* may define $Sum\ Int\ i = Zero$.

Thus we cannot complete proofs of lemmas that holds for *all* possible types. There are two ways to work around this difficulty: close the world, or leave it open but only to new instances that satisfy the lemma.

Closed world One possible solution is to introduce what we would call *datakinds*, which introduce new kinds, along with associated type constructors, much like *datatypes* introduce new types with associated data constructors. This is the approach taken in Omega [22] as well as in most proof assistants such as Coq [6] and Agda [1]. This would work well for our type-preserving compiler, but would be a significant departure from GHC's current type families.

Open world If the world is wide open, our lemma simply does not hold in general, so we have no hope of proving it. We have to close the world to some extent

but we can leave it ajar: rather than disallow extending the type family with new instances altogether, we will simply constrain new instances to obey the lemma(s) that apply to the type family.

This idea is somewhat similar to type class inheritance or to functional dependencies of multi-parameter type classes: functional dependencies also restrict the set of possible instances, so as the make sure that a property is preserved, and type class inheritance requires every instance to obey the constraints imposed by the parent.

4.6.1 Syntax

Ideally, we would like to state properties such as Lemma 4.1, and have the type checker verify that all instances of the relevant families satisfy the property. The syntax for introducing such lemmas could be as follows:

$$\textbf{lemma}\ \mathit{substCpsCommute} : \mathit{Subst}\ (\mathit{CPS}\ s)\ (\mathit{CPS}\ t)\ i \sim \mathit{CPS}\ (\mathit{Subst}\ s\ t\ i)$$
$$\textbf{lemma}\ \mathit{cpsInjective} : \mathit{CPS}\ a \sim \mathit{CPS}\ b \Rightarrow a \sim b$$

These two declarations would have the effect of introducing two functions (of the same name) which can be used to discharge the constraint expressed by each lemma:

$$\mathit{substCpsCommute} :: (\mathit{Subst}\ (\mathit{CPS}\ s)\ (\mathit{CPS}\ t)\ i \sim \mathit{CPS}\ (\mathit{Subst}\ s\ t\ i) \Rightarrow a) \rightarrow a$$
$$\mathit{cpsInjective} :: \mathit{CPS}\ a \sim \mathit{CPS}\ b \Rightarrow (a \sim b \Rightarrow t) \rightarrow t$$

Of course, this sort of identity function should be optimized out so as to have no run-time cost. Resuming our example from Sec. 4.5.1, we could write *cps* as:

$$\begin{array}{l} \mathit{cps}\ (\mathit{TpApp}\ e)\ k = \mathit{substCpsCommute}\ e' \\ \quad \textbf{where}\ e' :: \mathit{Subst}\ (\mathit{CPS}\ s)\ (\mathit{CPS}\ t)\ Z \sim \mathit{CPS}\ (\mathit{Subst}\ s\ t\ Z) \Rightarrow \mathit{ExpK} \\ \qquad\quad e' = \mathit{cps}\ e\ (\lambda x \rightarrow \mathit{TpAppK}\ x\ (\mathit{LamK}\ k)) \end{array}$$

or better yet:

$$\begin{array}{l} \mathit{cps}\ (\mathit{TpApp}\ e)\ k = \mathit{substCpsCommute}\ \mathit{cpsTpApp}\ e\ k \\ \quad \textbf{where}\ \mathit{cpsTpApp} :: \mathit{Subst}\ (\mathit{CPS}\ s)\ (\mathit{CPS}\ t)\ Z \sim \mathit{CPS}\ (\mathit{Subst}\ s\ t\ Z) \Rightarrow \\ \qquad\qquad\qquad\qquad \mathit{Exp}\ (\mathit{All}\ s) \rightarrow (\mathit{ValK}\ (\mathit{CPS}\ (\mathit{All}\ s)) \rightarrow \mathit{ExpK}) \rightarrow \mathit{ExpK} \\ \qquad\quad \mathit{cpsTpApp}\ e\ k = \mathit{cps}\ e\ (\lambda x \rightarrow \mathit{TpAppK}\ x\ (\mathit{LamK}\ k)) \end{array}$$

where the coercion *substCpsCommute cpsTpApp* can be lifted outside of the recursion, should it have a run-time cost.

Explicit proofs If fully automatic checking of the lemmas turns out to be impractical, we will need to provide explicit proofs along with instance declarations. If for example we extend *CPS* and *Subst* to handle product types, the syntax for proving that the new instances satisfy the lemma could look as follows:

type instance $CPS\ (a,b) = (CPS\ a,\ CPS\ b)$
type instance $Subst\ (a,b)\ t\ i = (Subst\ a\ t\ i,\ Subst\ b\ t\ i)$
proof *substCpsCommute* : $CPS\ (Subst\ (a,b)\ t\ i)$
$\{$- reduce -$\}$ $\sim (CPS\ (Subst\ a\ t\ i),\ CPS\ (Subst\ b\ t\ i))$
$\{$- induction -$\}$ $\sim (Subst\ (CPS\ a)\ (CPS\ t)\ i,\ Subst\ (CPS\ a)\ (CPS\ t)\ i)$
$\{$- reduce -$\}$ $\sim Subst\ (CPS\ (a,b))\ (CPS\ t)\ i$

The proof consists of a series of coercions that begins with the left-hand side of the coercion to prove valid and ends with its right-hand side. Each step of the proof is justified by either applying a known lemma, or simply because the two types reduce to the same canonical form.

As shown here, a particular lemma may depend on more than one type family. Also they will often need to resort to induction. Additionally to checking that the proof steps are correct, the system will need to verify that the induction, if any, is well-founded, that the various proof chunks provided do cover all (currently) possible cases. This last problem can be difficult in the presence of dependent types [20], but as long as Haskell's type systems remains itself simply typed, this should not be major hurdle.

4.7 RELATED WORK

Type classes were proposed in [25] and extended to multiple parameters with functional dependencies in [11]. GADTs have been proposed several times in one form or another and under a variety of names, see for example [5, 23, 26]. Type families were proposed in [18, 17] and are related to associated types [3].

Taming the open world assumption has already been done several times in different contexts. We have already mentioned that the inheritance hierarchy of type classes imposes constraints on the possible new instances [25]. And similarly functional dependencies [11], used to restrict the set of instances of some multi-parameter type classes, to make sure that although the world is open, the unknown part is constrained to obey the property described by the functional dependencies.

In a different context, Carsten Schürmann [19] uses a *regular world assumption* to constrain the type environments in LF judgments, so as to circumvent the difficulties inherent to the non-inductive nature of *higher-order abstract syntax*.

[21] shows what a CPS translation would look like with the equivalent of type families in a closed world. [14] presents a more sophisticated set of coercions where the coercions can be computed by type-level functions, thus allowing to write proofs of type-equivalence lemmas at the level of types.

The details of our experience writing a type preserving compiler in Haskell can be found in [7, 8, 9].

4.8 CONCLUSION

Results We have updated every phase of the compiler (CPS conversion, closure conversion, a function hoisting phase, and a conversion from higher-order abstract

syntax to de Bruijn indices) to use type families instead of GADTs in the way illustrated here, and the results have been largely positive, cutting down on code size drastically and improving performance significantly.

For the purpose of the comparison, we assembled two versions of the CPS conversion over a simply typed language (with integers, pairs and recursion), one using proof witnesses encoded as GADTs, and one using type functions. The use of type functions resulted in a speedup of an order of magnitude, in fact speeding up by a factor closer to 30 when compiled with GHC version 6.8.2. The size of the code implementing the transformation also dropped by roughly 40%.

Acknowledgments We'd like to thank Tom Schrijvers and Martin Sulzmann for fruitful discussions.

Source code The source code used for the comparison is available from the author's web page:

`http://www-etud.iro.umontreal.ca/~guillelj/cps-tf/`

REFERENCES

[1] The Agda programming language. http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php.

[2] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 2005. ACM Press.

[4] C. Chen, D. Zhu, and H. Xi. Implementing Cut Elimination: A Case Study of Simulating Dependent Types in Haskell. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages*, pages 239–254, Dallas, TX, June 2004. Springer-Verlag LNCS vol. 3057.

[5] J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[6] The Coq proof assistant. http://coq.inria.fr.

[7] L.-J. Guillemette and S. Monnier. Type-safe code transformations in Haskell. In *Programming Languages meets Program Verification*, volume 174(7) of *Electronic Notes in Theoretical Computer Science*, pages 23–39, Aug. 2006.

[8] L.-J. Guillemette and S. Monnier. A type-preserving closure conversion in haskell. In *Haskell Workshop*. ACM Press, Sept. 2007.

[9] L.-J. Guillemette and S. Monnier. A type-preserving compiler in haskell. In ICFP'08 [10].

[10] *International Conference on Functional Programming*, Victoria, BC, Sept. 2008.

[11] M. P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, volume 1782 of *LNCS*, pages 230–244, 2000.

[12] F. Kamareddine. Reviewing the classical and the de bruijn notation for λ-calculus and pure type systems. *Journal of Logic and Computation*, 11, 2001.

[13] C. McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(5):375–392, 2002.

[14] S. Monnier. The Swiss coercion. In *Programming Languages meets Program Verification*, pages 33–40, Freiburg, Germany, Sept. 2007. ACM Press.

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Symposium on Principles of Programming Languages*, pages 85–97, Jan. 1998.

[16] S. Peyton-Jones et al. The Haskell Prime Report. Working Draft, 2007.

[17] T. Schrijvers, S. Peyton Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In ICFP'08 [10].

[18] T. Schrijvers, M. Sulzmann, S. Peyton Jones, and M. M. T. Chakravarty. Towards open type functions for Haskell. Presented at IFL 2007, 2007.

[19] C. Schürmann. A meta logical framework based on realizability. In *Logical Frameworks and Meta-Languages*, Santa Barbara, CA, June 2000.

[20] C. Schürmann and F. Pfenning. A coverage checking algorithm for lf. In *International Conference on Theorem Proving in Higher-Order Logics*, Sept. 2003.

[21] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Symposium on Principles of Programming Languages*, pages 217–232, Jan. 2002.

[22] T. Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119, New York, NY, USA, 2004. ACM Press.

[23] T. Sheard and E. Pašalić. Meta-programming with built-in type equality. In *Logical Frameworks and Meta-Languages*, Cork, July 2004.

[24] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Types in Language Design and Implementation*, Jan. 2007.

[25] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Symposium on Principles of Programming Languages*, Austin, TX, Jan. 1989.

[26] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, LA, Jan. 2003.