Type Invariants for Haskell

Tom Schrijvers *
Katholieke Universiteit Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Louis-Julien Guillemette
Université de Montréal, Canada
guillelj@iro.umontreal.ca

Stefan Monnier
Université de Montréal, Canada
monnier@iro.umontreal.ca

Abstract

Multi-parameter type classes, functional dependencies, and recently GADTs and open type families open up opportunities to use complex type-level programming to let GHC's type checker verify various properties of your programs. But type-level code is special in that its correctness is crucial to the safety of the program; so except in those cases simple enough for the type checker to see trivially that the code is correct (or harmless), type-level programs need to come with a specification of their properties together with their proof.

In this article, we propose an extension to Haskell that allows the specification of invariants for type classes and open type families, together with accompanying evidence that those invariants hold. To accommodate the open nature of type classes and type families, the evidence itself needs to be open and every subcase of the proof can be provided independently from the others.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Functional Languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type Structure

General Terms Algorithms, Languages

Keywords Haskell, type checking, type functions, type families

1. Introduction

Multi-parameter type classes (Wadler and Blott 1989; Peyton Jones et al. 1997; Duggan and Ophel 2002), functional dependencies (Jones 2000), and recently GADTs (Xi et al. 2003; Cheney and Hinze 2003; Peyton Jones et al. 2006) and open type families (Schrijvers et al. 2008) open up opportunities to use complex type-level programming to let GHC's type checker verify various properties of your programs. But type-level code is special in that its correctness is crucial to the safety of the program; In surprisingly many cases the type checker's limited reasoning power is actually sufficient to verify completely automatically that the type-level code is correct. But in the general case, the programmer needs to help the type checker by instructing it to exploit some particular properties of the types manipulated.

As it happens, GHC's type system is already sufficiently powerful that in most if not all cases, the code can be rewritten in such a way that the type checker can check the correctness of the type annotations, even for fairly complex cases such as when proving that a compiler is type preserving (Guillemette and Monnier 2008). But such rewrites are unsatisfactory because they tend to either incur substantial run-time cost, or force wide-reaching changes which break the modularity of the code, or both.

So we basically want our type language to include a proof assistant. One alternative is to retrofit a system such as the Calculus of Inductive Constructions (Paulin-Mohring 1993) or Twelf (Pfenning and Schürmann 1999) into GHC's type language, but since GHC's type system is already fairly complex and powerful we decided instead to try to limit ourselves to a small extension of its type language, so as to make good use of the existing machinery. One of the side benefits is that the system arguably integrates better with the rest of Haskell and should hopefully be more palatable for Haskell coders.

Concretely, we propose an extension of GHC's type system which allows the programmer to specify on the one hand some type invariants and on the other the corresponding proof. GHC already supports some forms of type invariants: a type class declaration can specify that all instances of this class should also be instances of some other classes, and for multi-parameter classes it can also specify functional dependencies between the parameters. Our type invariants subsume those two particular cases, except for the fact that they require more annotations: being more general, not only does the programmer need to provide explicit proofs of those invariants, but she additionally needs to explicitly specify when the invariant is used. To this end, each invariant provides a corresponding coercion function.

Another difference with the functional dependencies and the class context invariants, is that our type invariants can be specified separately from any class declaration, and can also apply to a combination of several classes or type families. This makes it possible to retroactively specify that all `Num` instances are also instances of `Additive`, without having to change the `Num` class.

The specific contributions of this paper are:

- We clearly motivate the need for type invariants with examples (Section 2).
- We present type invariants for Haskell, and explain and illustrate the choices made in our approach (Section 3).
- The formalization of our approach (Section 4) respects as much as possible the open and modular nature of Haskell's type classes and type families.
- We provide an external proof language that infers proof steps and greatly simplifies writing proofs (Section 5.1).
- As experimental evaluation, we have written and checked a number of invariants and their proofs (Section 5.2).

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

At the end of this paper, we discuss related work (Section 6) and conclude (Section 7).

2. Motivation and Examples

There are many examples where type families would be much more useful if they would support additional invariants beyond the basic axioms, i.e. the type family instances.

2.1 Indexed List Processing

Parity Indexing Ki Yung Ahn proposed the following list GADT on the Haskell mailing list, which is *parity*-indexed, i.e. indexed by whether the length is odd or even:

```
data Even
data Odd

data List a l where
  Nil  :: List a Even
  Cons :: a -> List a l -> List a (Flip l)
```

For instance, `Cons True Nil :: List Bool Odd` and `Cons False (Cons True Nil) :: List Bool Even`. The type family `Flip` expresses what happens to the parity of the list length if we add an element:

```
type family Flip l

type instance Flip Even = Odd
type instance Flip Odd  = Even
```

While some functions using these lists are easy to type check, e.g.,

```
headList :: List a l -> a
headList (Cons x xs) = x
```

it is unfortunately not possible to consider some other basic functions as well-typed:

```
tailList :: List a l -> List a (Flip l)
tailList (Cons x xs) = xs
```

From the GADT pattern match we know that `Flip k ~ l`, where `k` is the parity of `xs`, and `~` is the intensional equality predicate. From the signature, we can also see that `Flip l ~ k`. Once we eliminate `k` from these equations, we end up with the need to prove the following constraint:

$$\forall \texttt{l}.\texttt{Flip (Flip l) ~ l}. \tag{1}$$

While we may think that this equation readily holds for any `l`, this is not so for 3 reasons: First, the $\forall$`t` quantifier really means any type, even if it is not in the domain of `Flip`, so the equation should hold also for `Flip (Flip Char) ~ Char`. Second, due to the openness of the type family `Flip`, one may add at any time an additional type instance, e.g.,

```
type instance Flip Char = Even
```

such that `Flip (Flip Char) = Odd` and Equation (1) does not hold for `Char`. Finally, even if we resolve those problems, Haskell still provides no way to ask the type checker to verify and use this property.

There are various ways to work around the problem, such as cluttering either the type signature of `tailList` or the `List` constructors themselves with the required equality. Yet, this defers the equality to elsewhere in the program. It would be much neater if Equation (1) could simply be enforced on all instances of the type family, and that subsequently we could simply use Equation (1) wherever needed.

In (Ahn and Sheard 2008) Ki Yung Ahn decided to use an inelegant encoding of the indexing types that avoids type families.

Length Indexing Length-indexed lists are a classical example:

```
data Z
data S n

data List a l where
  Nil  :: List a Z
  Cons :: a -> List a n -> List a (S n)
```

where we need a type family `Add` for expressing the signature of `append`:

```
append :: List a k -> List a l -> List a (Add k l)
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

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

The same type family would also serve us for the signature of `merge`:

```
merge :: List a k -> List a l -> List a (Add k l)
merge Nil         ys = ys
merge (Cons x xs) ys = Cons x (merge ys xs)
```

In the second clause, the GADT pattern match exposes the equality `k ~ S k'`, where `k'` is the length of `xs`. Now the type checker expects the type `Add k' l` for the expression `(merge ys xs)`. However, it infers, based on the signature of `merge`, the type `Add l k'`. Hence, for this code to type check, commutativity of `Add` must hold:

$$\forall \texttt{k}.\forall \texttt{l}.\texttt{Add k l ~ Add l k} \tag{2}$$

So again, we would want to be able to take advantage of invariants on type families.

Invariants as term-level functions Without support for type invariants, it is still possible to implement the invariants at the term level, so as to make the above code examples type-check. We first need to reify the equality predicate at the term level using a GADT which witnesses the equivalence of two types, as well as reify the types (such as the length annotations) using singleton types:

```
data Eqv s t where
  Eqv :: (s ~ t) => Eqv s t

data Nat n where
  Nz :: Nat Z
  Ns :: Nat s -> Nat (S s)
```

An invariant such as the commutativity of addition is then implemented as a function that analyzes the term-level representatives and constructs the proof:

```
comm :: Nat i -> Nat j -> Eqv (Add i j) (Add j i)
comm Nz Nz     = Eqv
comm Nz (Ns j) = case addZ j of Eqv -> Eqv
comm (Ns i) Nz = case addZ i of Eqv -> Eqv
comm (Ns i) (Ns j) =
  case comm i j of
    Eqv -> case (comm (Ns i) j, comm i (Ns j)) of
             (Eqv, Eqv) -> Eqv
```

where `addZ` similarly implements an auxiliary invariant stating that:

$$\forall \texttt{n}.\ \texttt{Add n Z ~ n}. \tag{3}$$

To apply the invariant and have the `merge` function type-check, the length of the lists must be computed separately:

```
merge (Cons x xs) ys =
  case comm (length xs) (length ys) of
    Eqv -> Cons x (merge ys xs)

length :: List a l -> Nat l
length = ...
```

Of course, implementing invariants in this way has a cost at runtime, and since Haskell does not enforce that such a "proof" covers all the cases and will not loop indefinitely, we would need to rely on an external verifier to check the totality of that function.

2.2 Type Preserving Compilation

In Guillemette and Monnier's type preserving compiler (2008), an abstract syntax tree for System F is type-indexed with the source-level types. Type functions are used to relate the source-level types before and after various program transformations, i.e. continuation-passing-style (CPS) transformation and type. One of the invariants that comes up in this context is the commutativity of CPS transformation and type substitution:

```
∀t,x,s.CPS (Subst t x s) ~ Subst (CPS t) x (CPS s)
```

where `CPS t` reflects the type of an expression of source type t after CPS transformation, and `Subst t x s` is the type after substitution of the type variable `x` with type `s`.

3. Outline of Our Approach

In this section, we outline our approach, what works and what doesn't, and a number of complications that arise.

3.1 Domain Restriction

It is quite clear early on that invariants are not intended to be instantiated at just any types. For instance, we do not wish the invariant (2) to be instantiated to `Add Z Char ~ Add Char Z`, as there is not even an instance of `Add` that tells us what to do with `Add Char Z`.

An extreme case is a type family `EMPTY` without any instance at all. Since there is no instance, we could readily assume that the two following invariants hold:

```
∀x. EMPTY x ~ Int
∀x. EMPTY x ~ Bool
```
(4)

In the absence of any instance, it is vacuously true that all instances of `EMPTY` satisfy the above two invariants. Yet, Haskell polymorphism is unconstrained, so the above ∀`x` does not restrict `x` to be a valid argument to `EMPTY`. So things can go very wrong if we do, as we can then derive `Int ~ EMPTY Char ~ Bool`. Type soundness is at stake!

We need a mechanism to restrict the use of invariants to *appropriate* types only. Perhaps the most obvious approach is to use a richer kinding system than the one Haskell offers where all proper types are of the same kind `*`. A richer kind system could allow the user to classify types into distinct kinds. For instance, analogous to Haskell data type definitions, we could specify that type-level natural numbers are of kind `Nat`:

```
datakind Nat = Z | S Nat
```

Then we could restrict the use of type variables in invariants to the appropriate kind:

```
∀k:Nat.∀l:Nat.Add k l ~ Add l k
```
(5)

The above `datakind` has two disadvantages: Firstly, it provides a closed definition of kinds whereas type classes and type families are open. Secondly and more importantly, it requires extending Haskell with another type system feature.

Instead we use a solution that takes advantage of an existing Haskell feature for classifying types and that is open, namely type classes. Instead of a datakind `Nat`, we propose to use a type class `Nat`:

```
class Nat n
instance          Nat Z
instance Nat n => Nat (S n)
```

Note that the type class does not need to have any methods; we are only interested in its type-level aspects. Using this type class, we can add the same style of constraint context to the invariant with which Haskell programmers are already familiar in function signatures.

```
∀k.∀l.(Nat k, Nat l) => Add k l ~ Add l k
```
(6)

3.2 Using Invariants

Due to their generality, invariants are unfortunately inappropriate for automatic use by the type checker algorithm, lest we give up on decidability of type checking. Firstly, they cannot be treated as type family instances, also called top-level equations in (Schrijvers et al. 2008), because the left-hand side is not necessarily in the proper *constructor form* to ensure either termination or confluence. Secondly, they cannot be treated as local equations, as provided in a function signature, because they contain schema (i.e. universally quantified) variables. In other words, and unsurprisingly, the completion algorithm of type families cannot be used to transform the invariants into a terminating and confluent rewrite system.

Because of the above issues, we propose that the programmer explicitly indicates when an invariant should be used. For this purpose the concrete syntax for invariant definitions introduces a name for the invariant. This then introduces a coercion function of the same name, which the programmer can use to apply the invariant where it is needed.

Example 1. The definition of the commutativity invariant in concrete syntax is:

```
type invariant add_comm =
    (Nat x, Nat y) => Add x y ~ Add y x
```

In return for this specification, we get a coercion function which has the signature:

```
add_comm :: (Nat x, Nat y)
         => (Add x y ~ Add y x => a)
         -> a
```

It is used as follows in the `merge` example:

```
merge (Cons x xs) ys =
    Cons x (add_comm (merge ys xs))
```

□

3.3 Proving Invariants

While type family instances are really the programmer-provided axioms of type families, invariants should be derivable from those axioms. An invariant which is not derivable is not a proper invariant, and may introduce unsoundness. Hence, to make sure an invariant actually holds, we require a proof.

Because of the proof obligation for invariants, we also call them *lemmas* throughout the rest of the paper. However, we must remember that invariants differ from lemmas on the account of openness. Whereas lemmas usually prove an existing property, our invariants also impose a property on future instances.

Case-based proofs Because of the open and modular nature of type classes and type families, we expect our proofs to be open and modular as well. Modularity of a proof means that it should be

based on case analysis where each case can be written separately. Openness means that when new instances are added, then also new cases of the proof can be added. Of course, in a (closed) program, the type checker should verify whether all cases are covered.

As we have already introduced type classes to restrict the instantiation of type variables in invariants, we will also use the type class instances to determine our proof cases.

Example 2. For the commutativity invariant we have the following four proof cases based on the Cartesian product of the `Nat` type class instances:

```
proofcase add_comm Z     Z     = ...
proofcase add_comm Z     (S m) = ...
proofcase add_comm (S n) Z     = ...
proofcase add_comm (S n) (S m) = ...
```

□

Proof steps Each proof case must provide evidence that a particular instance of the lemma holds. The consequent of the lemma being a type equivalence, we need to prove the two types equal. We can write such proofs as a sequence of steps from one type to the other where each step is either justified by the use of an invariant, or by the traditional rules of type equivalence built into Haskell's type system.

Example 3. The full first proof case of the commutativity axiom looks as follows:

```
proofcase add_comm Z Z = Add Z Z ~ Add Z Z.
```

This equation trivially holds by the usual type equivalence rules. The second proof case looks as follows:

```
proofcase add_comm Z (S m) =
     Add Z (S m) ~
     S (Add Z m) ~{ind add_comm}
     S (Add m Z) ~
     Add (S m) Z
```

The middle step is annotated with the type invariant it uses and the fact that it is an *ind*uctive step, while the first and last are left without such annotations since they only rely on the built-in type equivalence rules. □

Example 4. Recall the parity invariant:

```
type invariant parity p =
  Parity p => Flip (Flip p) ~ p
```

where

```
class Parity p

instance Parity Odd
instance Parity Even
```

The proof cases are trivial since they do not require the use of any type invariant:

```
proofcase parity Odd  = Flip (Flip Odd)  ~ Odd
proofcase parity Even = Flip (Flip Even) ~ Even
```

We could trivially allow such proof cases to be elided altogether, of course. □

Well-Founded Induction Many proofs over inductively defined types, such as the natural numbers, require induction themselves. E.g. in the proof cases of `add_comm` earlier, to show the commutativity invariant for the second case, `Z` and `S m`, we rely on the invariant to hold for `m` and `Z`.

The main concern with inductive proofs, using the invariant directly or indirectly in the proof, is whether the induction is well-founded. In order for it to be well-founded, the use of an invariant in a proof case should be strictly smaller than the case itself, according to some norm.

Example 5. In the previous example, the following norm can be used to establish the well-foundedness of induction:

$$\begin{aligned} |\texttt{add\_comm}\ x\ y| &= |x| + |y| \\ |\texttt{Z}| &= 0 \\ |\texttt{S}\ x| &= 1 + |x| \end{aligned}$$

So we have that:

$$\begin{aligned} & |\texttt{add\_comm}\ (\texttt{S}\ \texttt{n})\ \texttt{Z}| &= 1 + |\texttt{n}| \\ > \quad & |\texttt{add\_comm}\ \texttt{Z}\ \texttt{n}| &= |\texttt{n}| \end{aligned}$$

i.e., there is a proper decrease. □

We propose to use the same norm for checking well-foundedness of invariants as is used to enforce termination of type families. While this choice is restrictive, Haskell programmers are already familiar with it from type families, and it has not posed any difficulties for the proofs we have constructed so far.

A final issue is how to identify recursive uses of lemmas, and hence when to enforce the well-foundedness criterion. First, notice that any invariant used in a proof may potentially refer back to the invariant being proved, so unless we know which invariants are mutually recursive and which are not, we have to conservatively consider that all uses of invariants may be inductive and should hence use strictly smaller arguments.

It turns out that enforcing in all cases that all invariants are applied to strictly smaller arguments has proven to be overly restrictive. Hence, we require the uses of invariants in proofs to be annotated with whether they are non-recursive or (potentially) recursive (using the `ind` annotation). In the latter case, the well-foundedness criterion is (conservatively) enforced. In the former case, we have to check instead whether the annotation is justified. This requires a global inspection of the lemma dependency graph of a program, i.e. a non-modular whole-program check, to ensure that this use of a lemma indeed is not part of any cycle.

3.4 Invariants with Class

While all the examples we have shown so far involve invariants about type families, we also want to be able to state invariants of type classes. Part of the reason for it being that we sometimes need them in order to prove type family invariants.

Example 6. The proof of the axiom shown in Sec. 2.2 for the type-preserving compiler involves among others the following auxiliary invariant:

```
∀k.∀t.(Nat k, Type t) => U Z k t ~ t        (7)
```

Here, `U` is a type family defined such that `U i k t` increments all De Bruijn indices no smaller than `k` by `i` in type `t`. The invariant expresses that if the increment is zero, then the update has no effect on the type.

This auxiliary invariant is instantiated in another proof to:

```
∀t.Type t => U Z Z (CPS t) ~ CPS t          (8)
```

The use of this invariant is only justified if we can provide evidence for its context `(Nat Z, Type (CPS t))`. While the former component is trivial to show, we can only show the latter with the help of an auxiliary type class invariant of the form:

```
type invariant type_cps = Type t => Type (CPS t)
```

□

Type class invariants are also useful on their own: for enforcing dependencies between type class instances.

Example 7. For instance, we can retroactively specify that all `Num` instances are also instances of some new type class `Additive`, without having to change the `Num` class. This invariant is expressed

as:

$$\forall t.\texttt{Num t => Additive t} \quad (9)$$

□

It turns out that type class invariants are similar to equational invariants and they require similar proof cases based on the different possible instantiations of the context. The proof steps little by little transform one context into the other, and as before each step is either justified by the built-in type class rules of Haskell, or by the use of a type invariant which then needs to be mentioned in an annotation.

Example 8. The proof cases of the above `type_cps` invariant look like:

```
proofcase type_cps Int =
     Type Int => Type (CPS Int)
proofcase type_cps (a,b) =
     Type (a,b)          =>
     (Type a, Type b)    =>{ind type_cps}
     (Type (CPS a), Type (CPS b)) =>
     Type (CPS (a,b))
```

□

There is one notable difference between those two kinds of invariants: equality predicates only exist at the level of types and are completely erased at run-time, whereas type classes are generally reified as dictionaries at run-time. This implies that for an equational invariant the value-level coercion function can be trivially implemented as a no-op; whereas for class invariants the value-level coercion function needs to build the corresponding run-time evidence (i.e. dictionary).

4. Formalization of Invariants

In this section, we formalize the type-level fragment of our Haskell language extension. The proof cases written in the source program do not easily lend themselves to reasoning about properties such as soundness proofs. So the the proof cases are here mapped into a lower-level representation called type equality coercions, which are taken from System F_C (Sulzmann et al. 2007a), the intermediate language of GHC, with some simple extensions.

4.1 Coercions

Type equality coercions provide a coercion calculus with which we can build complex type equality proofs. For example, the proof that `Add Z Z` is equal to itself is represented by the coercion:

```
refl (Add Z Z)
```

The proof that `Flip (Flip Odd) = Odd` is written as follows:

```
(refl Flip) flip_odd ∘ flip_even
```

The two `Flip` type family instances, denoted respectively `flip_odd` and `flip_even`, are used here as axioms.
`flip_odd` says that `Flip Odd ~ Even`;
`refl Flip` says simply that `Flip ~ Flip`;
the application `(refl Flip) flip_odd` combines them to prove that `Flip (Flip Odd) ~ Flip Even`;
finally ∘ combines it transitively with `flip_even` to conclude that `Flip (Flip Odd) ~ Odd`.

We have had to extend the coercion calculus of System F_C with a few more coercions in order to be able to represent all our proofs.

4.1.1 Invariant Coercions

Proof steps that are justified by type invariants, such as inductive steps, use a new kind of coercion, similar to the axioms that refer to type family instances, but which instead refer to a type invariant. This is used for instance, in order to show the commutativity invariant for the second case, `Z` and `S m`, where we rely on the invariant to hold for `Z` and `m`. With these so-called lemma coercions we can build both inductive proofs and proofs involving auxiliary lemmas.

Example 9. The above mentioned second proof case of the commutativity lemma is represented as follows:

```
proofcase add_comm Z (S m) =
     (add_Z (S m)) ∘ ((refl S) (sym (add_Z m)))
     ∘ ((refl S) (rec add_comm Z m))
     ∘ (sym (add_S m Z)
```

Where `(add_Z m)` is the axiom stating that `Add Z m = m`;
the axiom `add_S m Z` states that `Add (S m) Z = S (Add m Z)`;
and `sym` applies the commutativity of equality.
The first line corresponds to the first step in the proof which says that `Add Z (S m) ~ S (Add Z m)`.
The second line instantiates recursively the invariant `add_comm` to get a proof that `Add Z m ~ Add m Z` which it then lifts to `S (Add Z m) ~ S (Add m Z)`.

□

4.1.2 Context Proofs

A second complication of invariant coercions is posed by the invariant context. E.g. the proof above is actually incomplete: the use of the invariant `(rec add_comm Z m)` is only valid if the invariant's context can be provided, which in this case means we need to show both that the predicates `Nat Z` and `Nat m` are true. More generally, the use of an invariant coercion must be justified by a proof that the context is indeed satisfied. This requires a second coercion language, not for equality constraints, but for type class constraints. This coercion language is also used for type class invariants.

This time we do not have any pre-existing adequate representation available in System F_C: type class constraints are traditionally desugared to value-level dictionaries. The proof language is present only implicitly as value-level functions for dictionary construction and super class dictionary extraction. Hence, we propose a new coercion language modeled after these value-level functions. It contains additional functionality, such as instance context selectors which are usually not available for dictionaries (although such selectors have been proposed (Schrijvers and Sulzmann 2008)) but do make sense at the type level.[1]

Example 10. Further extending the above example, we must provide a proof of the context for the inductive argument `add_comm Z m`. In particular, we must show that both `Nat Z` and `Nat m` hold.

A proof of the former is `env (Nat Z)`, which says the proof can be found in the proof case's context. Indeed, `Nat Z` is a precondition of the proof case.

A proof of the latter is `isel 1 (env (Nat (S m)))`. Here, we have an instance `Nat (S N)` from the proof case's context. From this we can show that `Nat m` holds based on the type class instance `instance Nat m => Nat (S m)`. The proof constructor `isel` i says to take the ith type class constraint from the context of a particular instance. In this case $i = 1$, and `isel 1` yields the desired proof for `Nat m`.

In summary, the complete coercion for the inductive use of `add_comm`, with the context proofs explicit is:

```
rec (env Nat Z, isel 1 (env Nat (S m)))
    => add_comm Z m
```

□

[1] Note that it critically relies on the non-overlap of type class instances.

s, t, u, v	types
γ	coercions
g	equational axioms
h	equational invariants
p	type class invariants
δ	type class evidence
θ	type variable substitution
I_{TF}	type family instance
D_{TC}	type class declarations
I_{TC}	type class instance
π_{TF}	equational invariant proof case
π_{TC}	type class invariant proof case

Figure 1. Meta-Variables

$$
\begin{array}{lcl}
\gamma & ::= & \texttt{refl}\ t \mid g\ \overline{t} \mid \texttt{sym}\ \gamma \mid \gamma_1 \circ \gamma_2 \mid F\ \overline{\gamma} \mid \gamma_1\ \gamma_2 \mid \texttt{decomp}_{T_i}\ \gamma \\
 & \mid & \texttt{norec}\ \overline{\delta}\ \texttt{=>}\ h\ \overline{t} \mid \texttt{rec}\ \overline{\delta}\ \texttt{=>}\ h\ \overline{t} \\
\delta & ::= & \texttt{env}\ C\ t \mid \texttt{inst}\ \overline{\delta}\ (C\ t) \mid \texttt{ssel}\ i\ \delta \mid \texttt{isel}\ i\ \delta \\
 & \mid & \texttt{co}\ (C\ t)\ \gamma \mid \texttt{norec}\ \overline{\delta}\ \texttt{=>}\ p\ \overline{t} \mid \texttt{rec}\ \overline{\delta}\ \texttt{=>}\ p\ \overline{t}
\end{array}
$$

Figure 2. Proof Syntax

4.2 Program Syntax

Figure 1 summarizes our meta-variable naming conventions for the various syntactical categories. A *type-level program*, denoted by $\mathbf{prog}(\overline{D_{TC}}, \overline{I_{TF}}, \overline{I_{TC}}, \overline{h}, \overline{p}, \overline{\pi_{TF}}, \overline{\pi_{TC}})$, consists of:

- type class declarations $\overline{D_{TC}}$ of the form
$$\texttt{class}\ (C_1\ t_1, \ldots, C_n\ t_n)\ \texttt{=>}\ C\ a$$
- type class instances $\overline{I_{TC}}$ of the form
$$\texttt{instance}\ (C_1\ t_1, \ldots, C_n\ t_n)\ \texttt{=>}\ C\ t$$
- type family instances$\overline{I_{TF}}$ of the form
$$g\ \overline{x} : s_1 \sim s_2$$
- equational invariants $\overline{h}$ of the form
$$h\ \overline{x} : (C_1\ y_1, \ldots, C_m\ y_m)\ \texttt{=>}\ s_1 \sim s_2$$
where $\overline{y} \subseteq \overline{x}$, $vars(s_1) \subseteq \overline{x}$ and $vars(s_2) \subseteq \overline{x}$,
- type class invariants $\overline{p}$ of the form
$$p\ \overline{x} : (C_1\ y_1, \ldots, C_m\ y_m)\ \texttt{=>}\ C\ t$$
where $\overline{y} \subseteq \overline{x}$ and $vars(t) \subseteq \overline{x}$,
- proof cases $\overline{\pi_{TF}}$ for equational invariants of the form
$$\texttt{proofcase}\ h\ \overline{t} = \gamma$$
- proof cases $\overline{\pi_{TC}}$ for type class invariants of the form.
$$\texttt{proofcase}\ p\ \overline{t} = \delta$$

The syntax for proofs is listed in Figure 2. Its meaning is explained shortly.

For reasons of simplicity we have restricted ourselves to single-parameter type classes. We believe that it is straightforward to extend this work to multi-parameter type classes. We have also omitted type family declarations, because these do not contribute any useful information.

For the confluence and termination conditions on both type class and type family instances we refer to respectively (Sulzmann et al. 2007b) and (Schrijvers et al. 2008).

4.3 Well-Typing

The main typing rule for type-level programs is:

$$\frac{\begin{array}{c}\overline{D_{TC}} \cup \overline{I_{TF}} \cup \overline{I_{TC}} \cup \overline{h} \cup \overline{p} \vdash h_{i_1} : \diamond \\ \overline{D_{TC}} \cup \overline{I_{TF}} \cup \overline{I_{TC}} \cup \overline{h} \cup \overline{p} \vdash p_{i_2} : \diamond \\ \overline{D_{TC}} \cup \overline{I_{TF}} \cup \overline{I_{TC}} \cup \overline{h} \cup \overline{p} \vdash \pi_{TF,i_3} : \diamond \\ \overline{D_{TC}} \cup \overline{I_{TF}} \cup \overline{I_{TC}} \cup \overline{h} \cup \overline{p} \vdash \pi_{TC,i_4} : \diamond\end{array}}{\vdash \mathbf{prog}(\overline{D_{TC}}, \overline{I_{TF}}, \overline{I_{TC}}, \overline{h}, \overline{p}, \overline{\pi_{TF}}, \overline{\pi_{TC}}) : \diamond}$$

It checks the well-typedness of proof cases, and their complete coverage of invariants. The former well-typing rules are covered by Figures 3 and 4 for equational and type class proof cases respectively. A proof case is well-typed iff its proof term is well-typed and is a proof for the invariant instance that the proof case claims to cover. The proof terms for equational proof cases are the coercions of System F_C, extended with invariant coercions (INVCO and RECINVCO). The proof term language for type class invariants is new. Its primitives reflect the common value-level functions for dictionary construction and super class selectors used in the dictionary-passing implementation of type classes. Non-standard are the (ISELTC) rule reflects the instance selector function recently proposed in (Schrijvers and Sulzmann 2008), and the (COTC) rule for coercing a type class constraint from applying to one type to another. Finally, the rules (INVTC) and (RECINVTC) cover the type class counterparts of invariant coercions.

The environment Γ consists of equational axioms and invariants, type class instances and invariants, as well as type class constraints available in the current context.

4.4 Completeness Check

The coverage relation $\sqsubseteq$ checks whether one set of cases is covered by another set. It is used in LEMCO and LEMTC to check the completeness of the invariant proofs, i.e. whether all the required proof cases are supplied.

DEFINITION 1. *We define $\sqsubseteq$ as follows:*

$$S_1 \sqsubseteq S_2 \stackrel{def}{=} \forall t \in S_1. \exists s \in S_2. \exists \theta. \theta(s) \equiv t$$

Note that this formulation allows for multiple required proof cases to be covered by one and the same supplied proof case. In this way, a combinatorial explosion of proof cases can often be avoided.

Example 11. One of the invariants in the type preserving compiler of (Guillemette and Monnier 2008), expresses that CPS transformation before and after type variable substitution yield the same result:

```
invariant k_s =
    (Tp s, Tp t, Nat i) =>
        K (Subst s t i) ~ Subst (K s) (K t) i
```

There are two instances of `Nat` and four of `Tp`, and hence $32 = 4 \times 4 \times 2$ required proof cases. Luckily, we can get away with writing only four proof cases, only instantiating `s` to a more specialized type. □

4.5 Well-Foundedness

In order to ensure the well-foundedness of (directly or mutually) inductive proofs, we require that (potentially) inductive uses of invariants in proofs are annotated as such, with the keyword `rec`. In order to keep the well-foundedness check simple and modular we require that each such invariant use is strictly smaller than the covered proof case it appears in according to a well-founded partial order denoted by $\prec$ in rules (RECINVCO) and (RECINVTC).

DEFINITION 2. *The well-founded partial order $\prec$ is defined as:*

$$h_1\ \overline{t} \prec h_2\ \overline{s} \stackrel{def}{=} |\overline{t}| < |\overline{s}| \wedge vars(\overline{t}) \trianglelefteq vars(\overline{s})$$

$$\text{(ProofCo)}\quad \frac{h\,\overline{x}:\ (\overline{C\ v})\ \texttt{=>}\ s \sim t \in \Gamma \qquad \Gamma \cup \{C\ \overline{[u/x]v}\}; h\,\overline{u} \vdash \gamma : \overline{[u/x]}s \sim \overline{[u/x]}t}{\Gamma \vdash \texttt{proofcase}\ h\,\overline{u} = \gamma : \diamond}$$

$$\text{(ReflCo)}\quad \Gamma \vdash \texttt{refl}\ t : t \sim t \qquad \text{(SymCo)}\quad \frac{\Gamma \vdash \gamma : s \sim t}{\Gamma \vdash \texttt{sym}\ \gamma : t \sim s}$$

$$\text{(TransCo)}\quad \frac{\Gamma \vdash \gamma_1 : t_1 \sim t_2 \quad \Gamma \vdash \gamma_2 : t_2 \sim t_3}{\Gamma \vdash \gamma_1 \circ \gamma_2 : t_1 \sim t_3}$$

$$\text{(TFAppCo)}\quad \frac{\Gamma \vdash \gamma_i : s_i \sim t_i \quad i = 1, ..., n}{\Gamma \vdash F\ \gamma_1 ... \gamma_2\ : F\ s_1 ... s_n \sim F\ t_1 ... t_n}$$

$$\text{(AppCo)}\quad \frac{\Gamma \vdash \gamma_1 : f_1 \sim f_2 \qquad \Gamma \vdash \gamma_2 : s_1 \sim s_2}{\Gamma \vdash \gamma_1\ \gamma_2 : f_1\ s_1 \sim f_2\ s_2}$$

$$\text{(DecompT)}\quad \frac{\Gamma \vdash \gamma : T\ s_1 ... s_n \sim T\ t_1 ... t_n}{\Gamma \vdash \texttt{decomp}_{T_i}\ \gamma : s_i \sim t_i} \quad (i \in 1..n)$$

$$\text{(AxCo)}\quad \frac{g\,\overline{x} : s_1 \sim s_2 \in \Gamma}{\Gamma \vdash g\,\overline{t} : \overline{[t/x]}s_1 \sim \overline{[t/x]}s_2}$$

$$\text{(InvCo)}\quad \frac{h\,\overline{x}:\ (\overline{C\ u})\ \texttt{=>}\ s \sim t \in \Gamma \qquad \Gamma \vdash \delta_i : C_i\ \overline{[t/x]}u_i}{\Gamma \vdash \texttt{norec}\ \overline{\delta}\ \texttt{=>}\ h\,\overline{t} : \overline{[t/x]}s \sim \overline{[t/x]}t}$$

$$\text{(RecInvCo)}\quad \frac{h\,\overline{x}:\ (\overline{C\ u})\ \texttt{=>}\ s \sim t \in \Gamma \qquad \Gamma \vdash \delta_i : C_i\ \overline{[t/x]}u_i \qquad h\,\overline{t} \prec h_{lhs}\ \overline{t_{lhs}}}{\Gamma; h_{lhs}\ \overline{t_{lhs}} \vdash \texttt{rec}\ \overline{\delta}\ \texttt{=>}\ h\,\overline{t} : \overline{[t/x]}s \sim \overline{[t/x]}t}$$

Figure 3. Type Equation Proof System

where the term norm $|\cdot|$ is defined as:

$$\begin{array}{rcl} |\overline{t}| & = & \sum_i |t_i| \\ |t_1\ t_2| & = & |t_1| + |t_2| \\ |T| & = & 1 \\ |F\ \overline{t}| & = & 1 + |\overline{t}| \\ |a| & = & 1 \end{array}$$

the function vars returns the multi-set (bag) of type variables:

$$\begin{array}{rcl} vars(\overline{t}) & = & \biguplus_i vars(t_i) \\ vars(t_1\ t_2) & = & vars(t_1) \uplus vars(t_2) \\ vars(T) & = & \emptyset \\ vars(F\ \overline{t}) & = & \biguplus_i vars(t_i) \\ vars(a) & = & \{a\} \end{array}$$

where $\uplus$ $\Subset$ are the multi-set union and subset relations, i.e. taking multiplicity into account.

Observe that this definition of $\prec$ considers induction on the combination of all invariant arguments.

$$\text{(ProofTC)}\quad \frac{p\,\overline{x}:\ (\overline{C\ v})\ \texttt{=>}\ C\ t \in \Gamma \qquad \Gamma \cup \{C\ \overline{[u/x]v}\} \vdash \delta : C\ \overline{[u/x]}t}{\Gamma \vdash \texttt{proofcase}\ p\,\overline{u} = \delta : \diamond}$$

$$\text{(EnvTC)}\quad \frac{C\ t \in \Gamma}{\Gamma \vdash \texttt{env}\ (C\ t) : C\ t}$$

$$\text{(InstTC)}\quad \frac{\Gamma \vdash \delta_i : C_i\ \theta(t_i) \qquad \texttt{instance}\ (C_1\ t_1, \ldots, C_n\ t_n)\ \texttt{=>}\ C\ t \in \Gamma}{\Gamma \vdash \texttt{inst}\ \overline{\delta}\ (C\ \theta(t)) : C\ \theta(t)}$$

$$\text{(SSelTC)}\quad \frac{\Gamma \vdash \delta : C\ \theta(a) \qquad \texttt{class}\ (C_1\ t_1, \ldots, C_n\ t_n)\ \texttt{=>}\ C\ a \in \Gamma}{\Gamma \vdash \texttt{ssel}\ i\ \delta : C_i\ \theta(t_i)}$$

$$\text{(ISelTC)}\quad \frac{\Gamma \vdash \delta : C\ \theta(t) \qquad \texttt{instance}\ (C_1\ t_1, \ldots, C_n\ t_n)\ \texttt{=>}\ C\ t \in \Gamma}{\Gamma \vdash \texttt{isel}\ i\ \delta : C_i\ \theta(t_i)}$$

$$\text{(CoTC)}\quad \frac{\Gamma \vdash \gamma : t_1 \sim t_2}{\Gamma \vdash \texttt{co}\ (C\ t_1)\ \gamma : C\ t_2}$$

$$\text{(InvTC)}\quad \frac{p\,\overline{x}:\ (\overline{C\ u})\ \texttt{=>}\ C\ v \in \Gamma \qquad \Gamma \vdash \delta_i : C_i\ \overline{[t/x]}u_i}{\Gamma \vdash \texttt{norec}\ \overline{\delta}\ \texttt{=>}\ p\,\overline{t} : C\ \overline{[t/x]}v}$$

$$\text{(RecInvTC)}\quad \frac{p\,\overline{x}:\ (\overline{C\ u})\ \texttt{=>}\ C\ v \in \Gamma \qquad \Gamma \vdash \delta_i : C_i\ \overline{[t/x]}u_i \qquad p\,\overline{t} \prec p_{lhs}\ \overline{t_{lhs}}}{\Gamma; p_{lhs}\ \overline{t_{lhs}} \vdash \texttt{rec}\ \overline{\delta}\ \texttt{=>}\ p\,\overline{t} : C\ \overline{[t/x]}v}$$

Figure 4. Type Class Proof System

$$\text{(LemCo)}\quad \frac{\begin{array}{c}\{(C_1\ u_1, \ldots, C_n\ u_n) \mid \\ (\texttt{instance}\ \ldots\ \texttt{=>}\ C_i\ u_i) \in \Gamma\ (1 \le i \le n)\} \\ \sqsubseteq \\ \{\overline{[v/x]}(C_1\ y_1, \ldots, C_n\ y_n) \mid \\ (\texttt{proofcase}\ h\,\overline{v} = \gamma) \in \Gamma\}\end{array}}{\Gamma \vdash h\,\overline{x}:\ (\overline{C\ y})\ \texttt{=>}\ s \sim t : \diamond}$$

$$\text{(LemTC)}\quad \frac{\begin{array}{c}\{(C_1\ u_1, \ldots, C_n\ u_n) \mid \\ (\texttt{instance}\ \ldots\ \texttt{=>}\ C_i\ u_i) \in \Gamma\ (1 \le i \le n)\} \\ \sqsubseteq \\ \{\overline{[v/x]}(C_1\ y_1, \ldots, C_n\ y_n) \mid \\ (\texttt{proofcase}\ p\,\overline{v} = \gamma) \in \Gamma\}\end{array}}{\Gamma \vdash p\,\overline{x}:\ (C_1\ y_1, \ldots, C_n\ y_n)\ \texttt{=>}\ C\ t : \diamond}$$

Figure 5. Complete Proofs

Example 12. Example 9, perhaps more clearly in the higher-level notation of Example 3, involves an inductive use `add_comm Z n` in the proof for `add_comm (S n) Z`.

We have:

$$\begin{array}{rclrcl} |\texttt{Z n}| & = & 2 & vars(\texttt{Z n}) & = & \{\texttt{n}\} \\ |\texttt{(S n) Z}| & = & 3 & vars(\texttt{(S n) Z}) & = & \{\texttt{n}\} \end{array}$$

Hence, the induction is well-founded as $2 < 3 \wedge \{\texttt{n}\} \trianglelefteq \{\texttt{n}\}$, i.e. `add_comm Z n` $\prec$ `add_comm (S n) Z`. □

The above condition is not imposed on invariant uses that are not annotated as potentially inductive. To do so would be overly restrictive and rule out proofs for many useful properties. This is safe only if the invariant uses is truly not inductive, and the annotation is not inadvertently or maliciously omitted by the programmer. Hence, we collect all edges in the invariant "call graph", and globally check whether no annotations have been omitted.

4.6 Type Soundness

The constructs we added to System F_C preserve its soundness. More specifically, the progress lemma still holds thanks to the completeness check: whenever we need to evaluate a call to a coercion function, the completeness check guarantees that the corresponding lemma does provide an applicable case. The type preservation lemma also still holds trivially. And the termination check is used to show that the coercion functions can indeed be implemented as no-ops.

5. Implementation and Evaluation

We have implemented a prototype well-typing checker in GHC-Haskell. This checker implements the rules formalized in Section 4. It is also able to reconstruct such proofs from the more compact and natural notation discussed used in the source language. We call the former the *internal proof language*, and the latter the *external proof language*.

5.1 The External Proof Language

A proof in the external proof language consists of a sequence of types $\tau_1 \sim \ldots \sim \tau_n$, and denotes a proof for the equation $\tau_1 \sim \tau_n$. Our checker contains an inferencer that reconstructs the internal language proof from this external language proof. Essentially, it reconstructs the internal proof for two subsequent types $\tau_i \sim \tau_{i+1}$ as a base case, and composes subsequent proofs γ_j and γ_{j+1} with the transitivity constructor (TRANSCO).

For the base case, a single external proof step $\tau_i \sim \tau_{i+1}$, the reconstruction currently proceeds top-down and only considers a finite number of possibilities:

- If τ_i and τ_{i+1} are identical, use the (REFLCO) rule.
- Otherwise:
  - repeatedly decompose the type equality into multiple proof obligations using the (APPCO) and (TFAPPCO) rules,
  - until the remaining proof obligations can be discharged with a single application of a rule from the set

    $\{(\text{REFLCO}), (\text{SYMCO}), (\text{AXCO}), (\text{INVCO}), (\text{RECINVCO})\}$

    whose (possible) subproofs all use the (REFLCO) rule only.

When using the (INVCO) or (RECINVCO) rule, also the context proof is reconstructed. For that purpose, all possible proofs with no more than three type class proof constructors are tried, with the exception of proofs involving the (COTC) rule.

This search is only done for prototyping purposes. We are planning a full implementation which integrates with GHC, where we will be able to reuse the typing machinery which already computes such coercions for type families and computes comparable coercions for type classes, so we will not need to resort to such arbitrary search depth limits.

5.2 Evaluation

As preliminary evaluation of the expressivity of our invariant language we have encoded all the invariants of Section 2 and their proof cases. Some statistics of these invariants are recorded in the table below:

main invariant	aux. invariants	cases	proof size	steps
parity	0 + 0	2 + 0	10	4
commutativity	0 + 0	4 + 0	60	19
CPS	5 + 1	21 + 3	327	101*

For each row the table lists the number of auxiliary invariants (equational + type class) the number of proof cases (equational + type class), and total proof size. As measure for proof size we use the number of coercion and type class proof constructors used. The last column denotes the number of proof steps written in the high-level notation, as a comparison to the low-level proof constructors.

There are 18 proof constructors not covered in $(*)$ because of current limitations of our naive proof search prototype. These comprise 11 constructors for the one type class invariant, and 7 constructors for the use of this type class invariant in the proof of an equational invariant.

The results indicate that our high-level notation is about three times more compact than the low-level one, which is a significant reduction in programmer effort. In addition to their compactness, we believe that high-level proofs can be written more quickly because the linear proof-style is much easier to grasp for the programmer.

The prototype implementation in Haskell is available at `http://www.cs.kuleuven.be/~toms/Haskell/`.

6. Related Work

The Chameleon system allows programmers to extend the type checker with additional Constraint Handling Rules (CHRs) (Stuckey and Sulzmann 2005). These are useful for encoding additional properties, but have a more operational flavor, being left-to-right rewrite rules. The CHRs are more expressive than our invariants in that they allow existentially quantified type variables in the right-hand side. Yet, Chameleon treats the CHRs as axioms, and leaves the responsibility for soundness, completeness and termination to the programmer.

Invariants on type-level functions is something that can be done naturally in Coq (Paulin-Mohring 1993), although its type functions are closed. To go open world, one possibility may be to use the type class library of (Sozeau and Oury 2008), but this will not work for invariants that link two type classes (like our type-preservation invariant links the two type families `CPS` and `Subst`), since the invariant and its cases belong in neither type class.

The Twelf theorem prover (Pfenning and Schürmann 1999) has type families defined under an open world assumption, but these type families define relations rather than functions. Being dependently typed, it can be used to prove arbitrary invariants involving type families. The proofs take a logic programming flavor, unlike the equational proof syntax proposed here. Twelf provides coverage and termination checking, and support for proof search.

Omega (Sheard 2004) provide type-level functions similar to type families except that they are closed. One can reason about types at the term level using GADTs, but there are no user-defined type classes, type invariants, or other support for type-level programs.

7. Conclusion & Future Work

We have shown the limitations of Haskell's current type language, comprising type classes and type families. To extend the expressivity, we have proposed *type invariants*, which respect the open nature of the aforementioned type system features. Our formalization takes care of soundness, completeness and well-foundedness for type invariants and their proofs.

There are many ways in which to improve our external proof language and the reconstruction of the internal proof, e.g. covering more substantial internal proofs, more efficient search, and interactive proving to name just a few. We plan to investigate which extensions are the most likely to alleviate the programmer's burden.

Acknowledgments

We are grateful to Brigitte Pientka, Martin Sulzmann and the anonymous reviewers for their helpful comments. Part of this work was conducted while the first author was a visitor at Université de Montréal.

References

Ki Yung Ahn and Tim Sheard. Shared subtypes: subtyping recursive parametrized algebraic data types. In *Haskell '08: Proceedings of the 1st ACM SIGPLAN Haskell symposium*, pages 75–86, New York, NY, USA, 2008. ACM.

James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

Dominic Duggan and John Ophel. Type-checking multi-parameter type classes. *J. Funct. Program.*, 12(2):133–158, 2002.

Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 75–86, New York, NY, USA, 2008. ACM.

Mark P. Jones. Type classes with functional dependencies. In *Proc. of ESOP 2000*, number 1782 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

Christine Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *International conference on Typed Lambda Calculi and Applications*. LNCS 664, Springer-Verlag, 1993.

Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: exploring the design space. In *Haskell Workshop*, Amsterdam, June 1997.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In *International Conference on Functional Programming*, Portland, Oregon, September 2006.

Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, July 1999.

Tom Schrijvers and Martin Sulzmann. Unified Type Checking for Type Classes and Type Functions, 2008. Poster at the International Conference on Functional Programming (ICFP'08).

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, New York, NY, USA, 2008. ACM.

Tim 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. ISBN 1-58113-833-4.

Matthieu Sozeau and Nicolas Oury. First-class type classes. In *21th International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. LNCS 5170, Springer-Verlag, 2008.

Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, 2007a.

Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. Understanding functional dependencies via Constraint Handling Rules. *J. Funct. Program.*, 17(1):83–129, 2007b.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Symposium on Principles of Programming Languages*, Austin, TX, January 1989.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, LA, January 2003.