Deconstraining DSLs

Will Jones Tony Field Tristan Allwood

Department of Computing
Imperial College London
{wlj05,ajf,tora}@doc.ic.ac.uk

Abstract

Strongly-typed functional languages provide a powerful framework for embedding Domain-Specific Languages (DSLs). However, building type-safe functions defined over an embedded DSL can introduce application-specific type constraints that end up being imposed on the DSL data types themselves. At best, these constraints are unwieldy and at worst they can limit the range of DSL expressions that can be built. We present a simple solution to this problem that allows application-specific constraints to be specified at the point of use of a DSL expression rather than when the DSL's embedding types are defined. Our solution applies equally to both tagged and tagless representations and, importantly, also works in the presence of higher-rank types.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Constraints, Data types and structures, Polymorphism; D.3.2 [*Language Classifications*]: Applicative (functional) languages

Keywords Static typing, constraints, domain-specific languages.

1. Introduction

Embedding a *Domain-Specific Language* (DSL) into a general-purpose programming language provides a simple and effective way to support domain-specific functionality without the need for a custom compiler [1, 14]. Modern functional languages provide particularly powerful host languages for DSLs due in part to their rich type systems which, appropriately exploited, can endow the DSL with important safety properties.

The usual way to enforce type-safety is to add constraints to the data types used to embed the DSL. However, these constraints do exactly what they say: they *constrain* the way the DSL can be used. This is fine when there is a single use, or implementation of the DSL, for example one involving the invocation of a domain-specific library or the generation of code for a specific hardware platform. However, if the intention is to support multiple implementations of the DSL, or to use a particular DSL expression in different ways, then these constraints can interfere in a way that precludes the co-existence of some (and in exceptional cases, all) implementations.

To illustrate the problem, consider the following (rather crude) Haskell embedding of a very tiny expression language comprising polymorphic values, conditionals and equality; we will refer to this throughout the paper:

```
data Exp a where
  ValueE :: a → Exp a
  CondE  :: Exp Bool → Exp a → Exp a → Exp a
  EqE    :: Eq a ⇒ Exp a → Exp a → Exp Bool
```

Here we have made use of *Generalised Algebraic Data Types* (GADTs) [24] as supported in the Glasgow Haskell Compiler (GHC). The intention is that the DSL be able to support any set of basic types, provided the arguments to EqE are expressions that are comparable under equality. The corresponding Eq a constraint is the *only* constraint that we want to impose on the DSL.

Let's now see what happens when we try to define a function over DSL expressions that imposes its own 'local' constraint. The function compileSM below implements a compiler for a very simple stack machine that has no support for floating-point arithmetic. To make the function type-safe it needs to enforce the rule that the DSL expression it is given contains only integers and booleans; the booleans False and True will be encoded as the integers 0 and 1 in the usual way. Ignoring unique label generation, we have:

```
compileSM :: Exp a → String
compileSM (ValueE x)
  = "PUSH " ++ show (toInt x) ++ "\n"
compileSM (CondE p t f)
  = compileSM p ++ "CMP #0\n" ++ "BEQ L1\n" ++
      compileSM t ++ "BR L2\n" ++
        "L1: " ++ compileSM f ++ "L2: "
compileSM (EqE e1 e2)
  = compileSM e1 ++ compileSM e2 ++ "EQ\n"
```

Here, toInt is overloaded to work only on Ints and Bools, which we can express using a type class, such as the following:

```
class IntBool a where
  toInt :: a → Int
instance IntBool Int  where toInt = id
instance IntBool Bool where toInt = fromEnum
```

The function fromEnum in Haskell's Enum class converts booleans to integers, as required.

Now we have a problem: compileSM doesn't type check because the use of toInt in the ValueE rule requires that a be a member of the IntBool class. A first thought might be to add a context to the type of compileSM itself to express the IntBool constraint:

```
compileSM :: IntBool a ⇒ Exp a → String
```

However, this doesn't solve the problem, because the a here refers only to the result type of the top-level expression. The Haskell type system has no way of proving that the arguments to EqE carry

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.
Copyright © 2012 ACM 978-1-4503-1054-3/12/09. . . $10.00

the same constraint: the a in EqE's type signature is *existentially* quantified.

The usual solution to this type of problem (see Axelsson et al. [2] and Chakravarty et al. [7] for examples) is to add a constraint to the type of ValueE as follows:

```
data Exp a where
  ValueE :: IntBool a ⇒ a → Exp a
```

compileSM now type checks, but the unfortunate effect of the constraint is to restrict *all* uses of the DSL to expressions containing only integers or booleans. For example, the expression ValueE sin, which would have been perfectly valid before we added the constraint, now fails to type check. This was clearly not the intention: what we wanted to do was restrict the set of DSL programs that are compilable with compileSM, not the DSL itself.

Any function over the DSL that imposes a similar constraint will compound the problem, as those constraints will need to be added to ValueE's type as well. In the worst case these constraints may collectively have no types in common, in which case it will be impossible to build a DSL expression at all! Although our motivation in this paper concerns DSLs this is in fact a problem with all data types, not just those that encode DSLs.

In this paper we present two solutions to this problem. The first is presented somewhat as an aside, as it requires nothing more than a relatively simple abstraction over the target implementation platform using similar techniques to that described by Hughes [11]. The problem with this is that it breaks down in the context of higher-rank types, which means that it becomes extremely cumbersome to define the type of a function that uses a given DSL expression in two *different* ways, as in:

```
f :: Exp p a → ...
f e = ... (compileSM e) ... (pretty e) ...
```

for example, where pretty pretty-prints a given expression. Assuming that p represents the target platform, typing this function requires instantiating p at the types required by compileSM and pretty simultaneously. Unfortunately, introducing the required higher-rank type *hides* the type p, preventing it from being used to constrain values dynamically. This is explained in more detail in Section 8; the reader might wish to read this section first before continuing with Section 2.

The second solution, which constitutes the main contribution of the paper, takes a completely different approach and solves the restriction problem even in the presence of higher-rank types. The idea is to replace concrete constraints in a data type with *generic constraints* which capture at the *type level* key properties of a DSL expression, such as the primitive object types that it depends upon or the set of DSL operations that are used in its construction. Independently of the DSL data type(s), the constraints imposed by individual functions over the data type are specified separately as *platform* constraints. The trick, which involves some quite subtle type-level reasoning, is to ensure that these platform constraints are compatible with the generic constraints of the given DSL expression at the point where the function is *applied*. This constitutes the main technical challenge of the paper.

The key principle that underpins our idea is that implementation-specific constraints should be imposed at the point of *use* of a data type, not at the point of *definition*, i.e. it embodies the established principle that an interface should be separated from its implementation(s).

Our contributions are as follows:

- We describe a method for imposing generic constraints on data types that avoids the restrictions ordinarily imposed by concrete constraints (Section 2). In the context of a DSL we show how these can be used to impose independent implementation-specific constraints, for example on the types (Section 2.3) and operations (Section 6) supported by a given target platform.
- We show how the idea, which is initially presented in the context of GADTs, can be applied equally to tagless representations (Section 5) and representations involving rank-2 types (Section 5.1).
- We show that both the compile-time and run-time overheads of the scheme are bounded by a constant that is a function of the size of the list of types forming the constraint set, which we expect typically to be small (Section 2.4).
- Our implementation makes use of many of the latest Haskell extensions, including kind polymorphism (Section 6), constraint kinds (Section 3) and promotion (Section 2) and thus serves to document the application of these extensions in a practical setting.

1.1 Motivation

This work has been inspired by our broader efforts to build type-safe embedded DSLs for exploiting heterogeneous multi-core parallelism. Our objective is to be able to target a multitude of platforms with a view to exploiting both control and data parallelism in a type-safe manner. From the typing perspective there are two key challenges: 1. Allowing a variety of platform-specific program analysis (performance modelling, optimisation etc.) and compilation functions to co-exist without restricting the application of the DSL; 2. Allowing the same DSL program to be compilable to multiple platforms in the same context. The latter is particularly important for heterogeneous data parallelism, for example. We do not discuss these issues in any detail in this paper, although the running examples we use throughout have been chosen to illustrate some of the problems that arise in this area.

2. Generic constraints

We now continue with the example of Section 1. The idea is to associate each value of type Exp with a list of types as which represents *symbolically* the types that appear in the expression. Individual, concrete, constraints (such as IntBool) are then replaced with a set of *generic* constraints of the form Elem a as (or, using infix notation, $a \in as$).

For the ValueE constructor, a single generic constraint $a \in as$ captures the notion that the type of object wrapped by ValueE must be an element of the list of types, as. Similarly for the CondE case, which also requires the constraint Bool $\in as$ to cater for the type of the predicate. The EqE case again follows suit, except that it also retains the original Eq a constraint. In many, if not all, cases it may suffice to generate these $(\cdot \in \cdot)$ constraints mechanically, e.g. using Template Haskell [26] (see Section 10.1), but for the time being we decorate the constructor types explicitly:

```
data Exp as a where
  ValueE :: (a ∈ as) ⇒ a → Exp as a
  EqE    :: (Eq a, a ∈ as, Bool ∈ as)
            ⇒ Exp as a → Exp as a → Exp as Bool
  CondE  :: (a ∈ as, Bool ∈ as)
            ⇒ Exp as Bool → Exp as a → Exp as a
            → Exp as a
```

Importantly, this definition will suffice for *all* functions defined over the DSL.

A point that is worth making here is that Haskell's type inference engine automatically deletes duplicate constraints. As an example, consider the following expression:

```
eMixed
    = CondE (EqE (ValueE (3 :: Int)) (ValueE (4 :: Int)))
        (ValueE (0.0 :: Float)) (ValueE (3.9 :: Float))
```

The constraint Bool $\in$ as is inferred twice: once when typing the subexpression EqE ... which computes the constraints (Eq Int, Int $\in$ as, Bool $\in$ as), and once when typing the conditional CondE ... which independently imposes the constraint Bool $\in$ as in addition to Float $\in$ as (the top-level result is an Exp as Float). The Eq Int constraint is trivially satisfied, and so is removed, and the duplicate Bool $\in$ as constraints are collapsed into one. The resulting inferred type for eMixed is thus:

$$(\mathsf{Int} \in as, \mathsf{Bool} \in as, \mathsf{Float} \in as) \Rightarrow \mathsf{Exp}\ as\ \mathsf{Float}$$

A key feature is that the constraints list explicitly the types that appear *within* an expression that are not visible simply by looking at the expression's top-level type (Exp as Float in the case of eMixed).

2.1 Aside: type-level lists

At first sight the idea of a list of *types* might seem rather odd, as lists in Haskell are traditionally data types (with kind $\star$). What we need here is the concept of a list itself being a type. Fortunately, a recent extension to Haskell [29] supports the *promotion* of certain values; in particular this allows Haskell's familiar list constructors to be used at the type level. The promotion of the Haskell list type produces:

- A *type* $'[]$ and a *type constructor* $(:)$ which may be used for building lists *at the type level*. The quote ($'$) in the name of $'[]$ exists to distinguish it from the traditional list type constructor $[]$ (of kind $\star \to \star$).
- A set of *kinds* $[\kappa]$, each of *sort* $\Box$ (pronounced 'box'). $'[]$ thus has kind $\forall \kappa.\ [\kappa]$ and $(:)$ has kind $\forall \kappa.\ \kappa \to [\kappa] \to [\kappa]$.

Note that our technique does not rely on Haskell's ability to support type-level lists. We could equally well use the 'pair' and 'unit' type constructors (as in HLIST [13]), for example (Int, (Bool, ())) instead of $'$[Int, Bool], but choose to take advantage of the rather more convenient list syntax and associated kind safety.

2.2 Picking types

At this point the list of types as is purely symbolic, so if an expression e is typed as $(\tau_1 \in as, \ldots, \tau_n \in as) \Rightarrow \mathsf{Exp}\ as\ \tau$ then the constraints state that as must contain *at least* the types $\tau_1, \ldots, \tau_n$ for e to have type Exp as τ, *whatever as happens to be*.

Given a constrained type, one thing we should certainly be able to do is *instantiate* some or all of the types specified in the constraints. Thus, in the same way that we can instantiate a in the type of Haskell's built-in function abs :: Num $a \Rightarrow a \to a$ to the specific instance abs :: Int $\to$ Int, for example, so we should be able to do the same for constraints involving $(\cdot \in \cdot)$. We would therefore like all of the following to be valid instantiations for the type of eMixed above:

```
eMixed :: (Bool ∈ as, Float ∈ as) ⇒ Exp (Int : as) Float
eMixed :: Exp '[Bool, Float, Int] Float
eMixed :: Exp '[Char, Bool, Float, Int] Float
```

whilst an attempted instantiation such as

```
eMixed :: Exp '[Bool, Int] Float
```

should fail to type check.

To see how this can be achieved we now develop the implementation of $(\cdot \in \cdot)$ and discuss some possible variations. We remark that, in what follows, considerable care is needed to ensure that we don't inadvertently restrict the ability of DSL designers to use Haskell's operator overloading; we elaborate on this in Section 7.

We have a basic requirement to ensure that a constraint $a \in as$ is only satisfied when the list of types as contains a. The usual approach when dealing with lists is to assert that if the head of as is a then the constraint is satisfied trivially and, if not, to seek evidence that a appears somewhere in the tail of as. The following GADT describes these two cases:

```
data Evidence a as where
  Head :: Evidence a (a : as)
  Tail  :: (a ∈ as) ⇒ Evidence a (b : as)
```

This states that if an object of type Evidence a as is Head then a is at the head of the as, and that if it is Tail then a *is* in the list, but is not at the head. Notice that the recursive check into the tail of the list is performed by virtue of the constraint $a \in as$ in the definition of Tail (but see also Section 2.2.1 below). Crucially, therefore, if a is *not* in as then it should be impossible to construct an object which matches the type of either Head or Tail. Thus, if an object has type Evidence a as then it is irrefutably the case that a is in as, for otherwise we have a type error! This is the key property that governs our definition of $(\cdot \in \cdot)$:

```
class a ∈ as where
  evidence :: Evidence a as
```

which can be read: 'a is an element of as and the value evidence is either Head or Tail, depending on where in the list a resides.' There are no other possibilities! Which of the two cases we have is now determined by the instance declarations:

```
instance a ∈ (a : as) where
  evidence = Head
instance (a ∈ as) ⇒ a ∈ (b : as) where
  evidence = Tail
```

Thus, for example, the following:

```
evidence :: Evidence Int '[Int, Char]
evidence :: Evidence Int '[Float, Int]
```

have values Head and Tail respectively, whereas there is no instance of $(\cdot \in \cdot)$ which provides evidence :: Evidence Int $'$[Bool].

Note that it is important that there is no case for $'[]$ because we want there to be no evidence that a is an element of $'[]$, which is surely a 'lie'! However, it is interesting to see what happens if we try to define such an instance. Rather conveniently, if we attempt the following:

```
instance a ∈ '[] where
  evidence = ...
```

then the only thing we can put on the right-hand side is $\perp$ (or error). Looking at this another way, such an instance can successfully encode the 'lie' that a is an element of $'[]$, but the body *cannot* provide the evidence!

2.2.1 Implementation note

A possible variation on the scheme proposed is to replace the constraint $a \in as$ in Tail with explicit evidence of the existence of a in the tail of as as follows:

```
Tail :: !(Evidence a as) → Evidence a (b : as)
instance (a ∈ as) ⇒ a ∈ (b : as) where
  evidence = Tail evidence    -- Note: the two evidences
                              -- have different types.
```

This works in principle, but we must ensure that the Tail constructor is strict in its argument (hence the '!' annotation) in order to force the recursion into the tail of as.

In both approaches, the two instances of $(\cdot \in \cdot)$ shown *overlap*, and may thus only be used when one is recognisably more specific than the other. This means that we may only eliminate $(\cdot \in \cdot)$ constraints when reasoning about lists of *ground* types: there is no way in general for the compiler to know if the type variable a is a member of the list $(b : as)$, i.e. whether a and b are the same type. This is not a serious limitation, however, as the constraints are typically only eliminated at the point where a concrete type must be picked anyway (see Section 10.2).

2.3 Platform constraints

Let us now return to the compileSM function for our simple stack machine. Thanks to compileSM's non-discriminating type, the EqE and CondE cases require no modification from those shown in Section 1. The ValueE clause is less obvious so let's reproduce it here:

```
compileSM (ValueE x)
  = "PUSH " ++ show (toInt x) ++ "\n"
```

From the definition of Exp above, all we know about the type of x (b, say) is that $b \in as$. For this particular implementation we need to establish the fact that b is also either an integer or a boolean, i.e. we need to satisfy the constraint IntBool b before we can invoke toInt.

The key is to look at the type of compileSM itself. There is clearly a *platform-specific* requirement that all types in a given expression must be either Ints or Bools so the place to specify this constraint must be in the definition of compileSM itself:

```
compileSM :: AllIntBool as ⇒ Exp a as → String
```

The idea is for the AllIntBool type class to define a 'wrapper' function, toInt', whose role is to pick the right instance of IntBool (here either the Int or Bool instance); this involves convincing the type checker that such an instance exists. In order to do this toInt' needs to carry with it the list of constrained types as:

```
class AllIntBool as where
  toInt' :: (b ∈ as) ⇒ Proxy as → b → Int
```

What is the role of the Proxy type? On the left of the $\Rightarrow$ we have the constraint that $b \in as$. To be able call toInt' we will need to pass an argument that makes the type as concrete, such that the Haskell compiler can pick the correct instance of AllIntBool. Usually this would involve passing an argument of type as, but as has kind $[\star]$ whereas function arguments must have kind $\star$. The Proxy data type suffices to effect the conversion:

```
data Proxy as = Proxy
```

Before we continue with AllIntBool, it will be useful to complete the compiler in order to see how the proxy value comes into play:

```
compileSM (ValueE x)
  = "PUSH " ++ show (toInt' (Proxy :: Proxy as) x)
      ++ "\n"
```

Let's now build the AllIntBool class instances, beginning with the instance for (:). If a is an instance of IntBool and the list as is AllIntBool then we need to extend the constraint to the list $(a : as)$:

```
instance (IntBool a, AllIntBool as)
      ⇒ AllIntBool (a : as) where
  toInt' _ x = ...
```

The trick is to observe the type of toInt':

```
     (IntBool a, AllIntBool as, b ∈ (a : as))
  ⇒  Proxy (a : as) → b → Int
```

Here, both a and b are *type variables* so in the event that b is at the *head* of the list $(a : as)$, we shall have that $a \equiv b$ and hence IntBool b. In this case, we can therefore apply toInt to x and we are done. If b is in the *tail* of the list $(a : as)$ then we require a proof that $b \in as$, in which case we will proceed by recursion. How do we know whether b is at the head of as? We use the evidence method of the corresponding $(\cdot \in \cdot)$ instance:

```
instance (IntBool a, AllIntBool as)
      ⇒ AllIntBool (a : as) where
  toInt' _ (x :: b)
    = case evidence :: Evidence b (a : as) of
        Head → toInt x
        Tail  → toInt' (Proxy :: Proxy as) x
```

Notice that in the recursive call to toInt' we must reconstruct a new proxy object, as its type must reflect that of the tail of $(a : as)$. This proxy argument is never referred to at the object level; its role is simply to pick the required type instance for AllIntBool.

An instance for $'[\,]$ is needed because the AllIntBool instance for $a : '[\,]$ (see above) requires the two constraints IntBool a and AllIntBool $'[\,]$. Of course, in this case it *must* be the case that the corresponding evidence is Head for otherwise the type we are looking for would not be in the list; a recursive call to toInt' with argument (Proxy :: Proxy $'[\,]$) can therefore never occur. We thus need the $'[\,]$ instance, but not its associated implementation of toInt':

```
instance AllIntBool '[] where
  toInt' _ (x :: b)
    = seq (evidence :: Evidence b '[]) ⊥
```

Note that we could simply have made the right-hand side $\bot$ but we instead 'call the bluff' of any instance of the form $b \in '[\,]$. We have seen earlier (Section 2.2) that the value of any alleged evidence can only be $\bot$; the above code thus explicitly exposes the lie!

2.4 Implementation cost

The process of picking concrete types for each $(\cdot \in \cdot)$ constraint (Section 2.2) incurs a compile-time overhead, as we must ensure that each $(\cdot \in \cdot)$ constraint is satisfied by the given list of concrete types. For example, for the DSL expression e with type $(\tau_1 \in as, \ldots, \tau_n \in as) \Rightarrow$ Exp as a we can impose any type of the form $e ::$ Exp τs a provided each τ_i appears in τs. If τs contains m types then the cost of the static type check is $O(mn)$.

There is also a run-time overhead, however, which depends on how the type classes involved are implemented. In what follows we shall assume the use of a *dictionary transformation* as presented by Wadler and Blott [28] to explain in principle what must happen at run time. Each overloaded function used in a given implementation (e.g. toInt in compileSM) must be invoked at the correct type. This is *not* done by traversing a list of types, however, as no type information is retained at run time. Instead, each constraint in a function's type (e.g. $\mathcal{C} \Rightarrow \ldots$) is translated by the Haskell compiler into an additional argument ($\mathcal{C} \to \ldots$) that implements a *dictionary* of functions that correspond to a specific instance of the corresponding class (here $\mathcal{C}$). In our example above, the invocation of the functions toInt', evidence and the pattern match on Tail at different types results in the introduction of different dictionaries at runtime. When evidence (whose dictionary corresponds to the constraint $b \in (a : as)$) returns Head the dictionary argument of toInt' so happens, by construction, to contain the required instance of IntBool, i.e. the instance for either Int or Bool.

It is not necessary to understand how the dictionary transformation effects this at run time (see the work of Wadler and Blott [28] for full details), but suffice it to say that the cost of invoking toInt at type τ in the example above is linearly proportional to the index, n say, of τ in the list τs in the imposed type Exp τs a. In short, we do not end up ‘searching’ for the correct dictionary for τ at run time; instead toInt' is invoked exactly n times whereupon its additional dictionary argument provides the required τ instance of the IntBool class.

2.5 Single expression, multiple use

Let us now consider what happens if we wish to apply several constrained functions to a single DSL expression. Since we have so far only seen the compileSM function, let us introduce a similarly constrained pretty-printer:

```
pretty :: AllShowable as ⇒ Exp as a → String
```

Here the constraint AllShowable as provides a guarantee that every type in as has a textual representation. We might apply both compileSM and pretty to a single expression, for example:

```
f e = ... (compileSM e) ... (pretty e) ...
```

For this to type check, e must be an expression of type Exp as a. The use of compileSM requires the constraint AllIntBool as to be satisfied. Additionally, the use of pretty introduces the constraint AllShowable as. Assuming that no other context is required, f's type will thus be inferred as:

```
f :: (AllIntBool as, AllShowable as) ⇒ Exp as a → ...
```

Note that e's type, Exp as a, makes no reference to either platform – it is completely removed from any implementation. A corollary of this is that e's type is the same at the call sites of compileSM and pretty. If it were not, e would have to be defined as a *polymorphic* argument and f would require a rank-2 type [23, 27]. We shall see later in Section 8 some of the problems that rank-2 types can create, all of which we avoid. Furthermore, we show in Section 5.1 that even when rank-2 types must be introduced, the flexibility of our terms remains unaffected.

3. Higher-order constraints

While AllIntBool is a useful type class, its implementation is both non-trivial and tied to the needs of the compileSM function. Far better would be to write one type class which encapsulates the notion of traversing type-level lists and subsequently *parameterise* it by a particular platform-dependent constraint. Thanks to GHC's recently-added support for *constraint kinds* (based on the work of Orchard and Schrijvers [19]), we can do just that:

```
class All c as where
  ...
```

All's first parameter, c, is a *constraint constructor*. As an example, the idea is that instantiating c to be IntBool will recreate the definition of AllIntBool above. Of course, this will only be the case if toInt' is generalised appropriately. To this end, let's attempt to define a function withElem, say, that abstracts over the result type (Int, in the case of toInt' above) , which must now carry the constraint c b:

```
class All c as where
  withElem :: (b ∈ as) ⇒ Proxy as → (c b ⇒ d) → d
```

Unfortunately, while this is a valid type in the eyes of the type checker,[1] this leads to ambiguous type constraints because of the fact that constraints in Haskell ‘float’ to the leftmost position in a type. As an example, consider the toInt function above. Its type is IntBool $b \Rightarrow b \rightarrow$ Int, a seemingly perfect fit for the above, which we would like to lead to the type:

```
withElem :: (b ∈ as) ⇒ Proxy as
            → (IntBool b ⇒ b → Int) → b → Int
```

However, the IntBool constraint is instead floated out to yield:

```
   (b ∈ as, IntBool b)
⇒  Proxy as → (c b ⇒ b → Int) → (b → Int)
```

which leaves an ambiguous constraint c b which the compiler is unable to solve. To mitigate this issue, we transform the constraint c b into a data type which ‘traps’ the relationship between c and b:

```
data Trap c b where
  Trap :: c b ⇒ Trap c b
```

Pattern matching on a constructor Trap of type Trap c b will bring into scope the constraint c b; similarly one may not create a value of type Trap c b unless the constraint c b is satisfiable. Thus we arrive at:

```
class All c as where
  withElem :: (b ∈ as) ⇒ Proxy as → (Trap c b → d) → d
```

The instances of All are now simple generalisations of the AllIntBool class above:

```
instance All c '[] where
  withElem _ (f :: Trap c b → d)
     = seq (evidence :: Evidence b '[]) ⊥
instance (c a, All c as) ⇒ All c (a : as) where
  withElem _ (f :: Trap c b → d)
     = case evidence :: Evidence b (a : as) of
          Head → f Trap
          Tail  → withElem (Proxy :: Proxy as) f
```

Note that the second instance requires Haskell's support for *undecidable instances* [22] as the type system cannot decide whether the constraint expressions c a and All c $(a : as)$ can ever be the same, in which case the type checker will loop. In this case it is easy to see that no instantiation of c can ever satisfy this property, not least because c a makes no reference to as.

To illustrate the use of withElem let's see how the compileSM function above can be defined in terms of an All c as constraint. In this case, withElem's second argument is a function of type Trap IntBool b $\rightarrow$ String and its job is to show the integer representation of a given b. The constraint that b is an instance of IntBool is now captured by a Trap value of type Trap IntBool b which must be named explicitly in a type signature:

```
compileSM :: All IntBool as ⇒ Exp as a → String
compileSM (ValueE x)
   = "PUSH " ++ withElem (Proxy :: Proxy as) (showInt x)
        ++ "\n"
     where
       showInt :: b → Trap IntBool b → String
       showInt x Trap = show (toInt x)
```

The other two cases are unchanged. Without the explicit type signature, the type of showInt would be inferred as:

```
showInt :: IntBool b ⇒ b → Trap c b → String
```

where the dictionary has an unconstrained (polymorphic) type. We must instead enforce the IntBool constraint on the dictionary itself; in short, we want a specific instance of showInt's principal type that enforces the constraint relationship between IntBool and b.

4. Summary

The basic apparatus we need is now in place, so this is a good point at which to summarise. We have shown that we can shift

[1] Provided that higher-rank types are permitted.

platform-specific type constraints from a DSL's embedding data type to the corresponding function(s) over that data type using the steps outlined below. At this point we'll assume we're working with tagged representations using GADTs, but we'll see shortly (Section 5) how the approach generalises to tagless representations.

1. Add generic constraints to each constructor's type signature that collectively identify the minimal set of types that must be supported on a given platform in order for the constructor to be used. For example, for a GADT $\mathcal{T}$ with constructor $\mathcal{C}$:

   ```
   data 𝒯 as a where
     𝒞 :: (Eq a, a ∈ as, Char ∈ as, ..., Int ∈ as)
        ⇒ 𝒯 as a → 𝒯 as Char → ··· → 𝒯 as Int
   ```

   where as represents *symbolically* a list of types. The key point is that these generic constraints enumerate not only the basic constraints such as Eq a but also constraints on the result type (here Int) and, importantly, existential types that are otherwise invisible outside the type (here a and Char).

2. Make each overloaded function f for which there is a platform-specific implementation a member of some type class $\mathcal{P}$ that encodes the platform constraint, for example:

   ```
   class 𝒫 a where f :: a → ...
   ```

   and define appropriate instances for each platform.

3. For a platform function p over the DSL data type $\mathcal{T}$ use higher-order constraint classes such as All to specify the type restrictions imposed on as by p, for example:

   ```
   p :: All 𝒫 as ⇒ 𝒯 as a → ...
   ```

4. Apply the platform function to a given DSL expression, e, say, by picking a specific type list as, for example,

   ```
   p (e :: 𝒯 '[Int, Char, ...] a)
   ```

   This imposes the platform-specific constraints on e, rather than $\mathcal{T}$. Type-safety is ensured by virtue of the fact that the chosen as must be compatible both with the generic constraints in e's type and the platform constraints encoded in p's type in order for the application to type check.

5. Tagless representations

Up until now, we have focused on a GADT-based representation of DSLs. Tagless representations [5, 21] offer an alternative approach to embedding a domain-specific language using functions rather than data constructors. To illustrate this, let's construct a tagless encoding of our simple expression language using generic constraints:

```
class TaglessExp e where
  valueE :: (a ∈ as) ⇒ a → e as a
  eqE    :: (Eq a, a ∈ as, Bool ∈ as)
          ⇒ e as a → e as a → e as Bool
  condE  :: (a ∈ as, Bool ∈ as)
          ⇒ e as Bool → e as a → e as a
          → e as a
```

Functions over terms in this tagless representation are realised as *instances* of the TaglessExp type class. As an example, the compileSM function above that was previously defined over a GADT must now be recast in terms of a data type and a corresponding TaglessExp instance that defines the valueE, eqE and condE functions. The data type is straightforward and may make use of All:

```
newtype CompileSM as a
  = CompileSM (All IntBool as ⇒ String)
```

There is a small complication with the definition of valueE, which can be seen when we try to define the TaglessExp instance:

```
instance TaglessExp CompileSM where
  valueE x
    = ...
```

At this point we need to construct a Proxy of type Proxy as, but we have no as to refer to. The solution is to define a helper function valueE', whose type signature reaffirms the constraint $a \in as$ on x's type:

```
valueE' :: (a ∈ as) ⇒ a → CompileSM as a
valueE' x
  = CompileSM
      ("PUSH " ++ withElem (Proxy :: Proxy as) (showInt x)
         ++ "\n")
```

showInt is as defined earlier. We can now complete the instance:

```
instance TaglessExp CompileSM where
  valueE x
    = valueE' x
  eqE (CompileSM s₁) (CompileSM s₂)
    = CompileSM (s₁ ++ s₂ ++ "EQ\n")
  condE (CompileSM p) (CompileSM t) (CompileSM f)
    = CompileSM $
        p ++ "CMP #0\n" ++ "BEQ L1\n" ++ t ++
          "BR L2\n" ++ "L1: " ++ f ++ "L2 :"
```

A function for compiling an expression to our stack machine can now be built simply by picking the correct instance of the TaglessExp class:

```
compileSM :: All IntBool as ⇒ CompileSM as a → String
compileSM (CompileSM s) = s
```

5.1 Rank-2 types

In contrast with GADT representations, tagless encodings are by construction parameterised by the implementation they target (e). Consequently, invoking multiple implementations of a single expression is not possible with the definitions given so far. Assuming the presence of Pretty, a tagless implementation of a pretty-printer, we may revisit our earlier example:

```
f e = ... (compileSM e) ... (pretty e) ...
```

f will not type check. To see why, assume that the type checker reaches compileSM first, whereupon the type variable e will be instantiated (via unification) to the type CompileSM, associated with the compiler. When the type checker later reaches the application of pretty, the checker will attempt a similar thing for the pretty printer, but will now attempt to unify the types CompileSM (the instantiation of e) and Pretty and that unification will fail. This is the classic limitation of rank-1 types: the type variable e can be instantiated to any type for which there is a corresponding TaglessExp instance, but it cannot be changed once it has been picked.

We may alleviate this problem by giving f a rank-2 type [23, 27], but better would be to close our terms once and for all:

```
newtype AnyTaglessExp as a
  = AnyTaglessExp (∀e. TaglessExp e ⇒ e as a)
```

f may now be rewritten to accept a value of type AnyTaglessExp:

```
f (AnyTaglessExp e) = ... (compileSM e) ... (pretty e) ...
```

A key point to note is that the type being constrained, as, 'escapes' the rank-2 type of AnyTaglessExp. In effect, AnyTaglessExp is equivalent to the Exp GADT introduced earlier.

6. Exploiting kind polymorphism: restricting operations

The previous sections have demonstrated how we may restrict the types of values that are introduced into a computation. We shall see now that we may also bound the *operations* that are used in an expression, thanks in part to GHC's support for *kind polymorphism* [29].

Thus far, the $(\cdot \in \cdot)$ and All type classes have been used to constrain lists of kind $[\star]$. However, their definitions afford them the following, more general kinds (note that Constraint is the kind of type class constraints – Show a or Eq b, for example):

```
(· ∈ ·) :: ∀κ. κ → [κ] → Constraint
All     :: ∀κ. (κ → Constraint) → [κ] → Constraint
```

Here, as one might expect, κ may be instantiated to *any kind*. What does this buy? Consider extending Exp to record the operations, os say, in addition to the types, that are used in the construction of an expression:

```
data Exp as os a where
  ...
```

We would like to use the $(\cdot \in \cdot)$ type class to constrain os, similar to the way we constrained types using as. We can achieve this by promoting the constructors of a data type such as:

```
data Op = EqOp | CondOp
```

to the type level. The modifications required to Exp are straightforward:

```
data Exp as os a where
  ValueE :: (a ∈ as) ⇒ a → Exp as os a
  EqE    :: (Eq a, a ∈ as, Bool ∈ as, EqOp ∈ os)
         ⇒ Exp as os a → Exp as os a → Exp as os Bool
  CondE  :: (a ∈ as, Bool ∈ as, CondOp ∈ os)
         ⇒ Exp as os Bool → Exp as os a → Exp as os a
         → Exp as os a
```

Note: we would ideally like to be able to promote the consructors of the Exp type itself, as in:

```
data Exp as os a where
  ...
  EqE :: (Eq a, a ∈ as, Bool ∈ as, EqE ∈ os)
      ⇒ Exp as os a → Exp as os a → Exp as os Bool
```

for example. However, Haskell does not allow the promotion of GADTs, as that would require coercions between kinds rather than the rather simpler notion of α-equivalence [29].

Given the modified version of Exp above, we may now pick and choose which operations are permitted in a given implementation. As an example, suppose we wish to generate code for an architecture in which conditional branching is undesirable (Nvidia's CUDA platform, for example). Once again, a type class may be defined to capture the operations that are supported:

```
class CUDACompatible o
instance CUDACompatible EqOp
```

The extension of CUDACompatible over a list os is then handled by the All class:

```
compileCUDA :: All CUDACompatible os
            ⇒ Exp as os a → String
compileCUDA
  = ...
```

(We omit the details of a full CUDA compiler!)

6.1 Combining types and operations

Parameterising Exp by both the types it contains (as) and the operations it uses (os) seems a little cumbersome. Another approach is to use some form of union operation, but at the type level. For example, we can use a promoted version of Haskell's Either type:

```
data Either a b = Left a | Right b
```

With this, Exp can instead be parameterised by a single list, ts, of kind $[\text{Either} \star \text{Op}]$; each item of ts will be *either* a type *or* an operation:

```
data Exp ts a where
  ValueE :: (Left a ∈ ts) ⇒ a → Exp ts a
  EqE    :: (Eq a, Left a ∈ ts, Left Bool ∈ ts,
              Right EqOp ∈ ts)
         ⇒ Exp ts a → Exp ts a → Exp ts Bool
  CondE  :: (Left a ∈ ts, Left Bool ∈ ts,
              Right CondOp ∈ ts)
         ⇒ Exp ts Bool → Exp ts a → Exp ts a
         → Exp ts a
```

However, the All family of type classes introduced in Section 3 doesn't fit well with this because All constrains *all* items in a list, whereas we need to be able to constrain either the as (Left) or the os (Right). We must therefore adapt the class by decomposing it into a pair of classes, each designed to constrain one of the types present in the list:

```
class AllLeft c ts where
  withLeftElem  :: (Left a ∈ ts)
                ⇒ Proxy ts → (Trap c a → d) → d
class AllRight c ts where
  withRightElem :: (Right b ∈ ts)
                ⇒ Proxy ts → (Trap c b → d) → d
```

If we consider the AllLeft class (the workings of AllRight follow suit), we see that we now need *two* instances for (:) that will begin:

```
instance (c a, AllLeft c ts)
      ⇒ AllLeft c (Left a : ts) where
  ...
instance AllLeft c ts ⇒ AllLeft c (Right a : ts) where
  ...
```

In the first case a type Left a can only be added to the list ts if a satisfies the constraint c. The instance is thus similar to that given in the definition of All:

```
instance (c a, AllLeft c ts)
      ⇒ AllLeft c (Left a : ts) where
  withLeftElem _ (f :: Trap c b → d)
    = case evidence :: Evidence (Left b) (Left a : ts) of
        Head → f Trap
        Tail → withLeftElem (Proxy :: Proxy ts) f
```

As for the second instance, it is in fact simpler – types of the form Right a may be added to the list ts regardless:

```
instance AllLeft c ts ⇒ AllLeft c (Right a : ts) where
  withLeftElem _ (f :: Trap c b → d)
    = case evidence :: Evidence (Left b) (Right a : ts) of
        Tail → withLeftElem (Proxy :: Proxy ts) f
```

Despite the absence of a Head clause in the **case**-statement, this function is total so an attempt to pattern match on Head will be a type error: the compiler knows that the types Left b and Right a will never be unifiable (a fact which Head would contradict).

With these instances in place, we can now reason about types and operations simultaneously. For example, if we have a CUDA-capable GPU which does not support double-precision arithmetic, we might type its compiler thus:

```
compileCUDASP :: (AllLeft SinglePrecision ts,
                  AllRight CUDACompatible ts)
              ⇒ Exp ts as → String
```

Of course, we can generalise this still, to produce a type such as:

```
compileCUDASP :: (All Left SinglePrecision ts,
                  All Right CUDACompatible ts)
              ⇒ Exp ts as → String
```

in which there is once again a single All class. We omit the details of its implementation.

7. Operator overloading

The lists of types discussed in this paper are not so much built as *described* – the presence of a list as is established before being appropriately constrained. A side-effect of this is that every subexpression in a term will be parameterised by the same list as – only the constraints may differ at any given point. Importantly, this is precisely what is required to overload many of Haskell's operators to work with Exp values. As an example, consider the Num type class:

```
class Num a where
  (+), (−), (∗) :: a → a → a
  abs, signum   :: a → a
  fromInteger   :: Integer → a
```

Suppose that we wish to overload the $(+)$ operator on expressions in our DSL. To support this we'll need a new Exp constructor which, in the 'vanilla' GADT of Section 2 might look like this:

```
data Exp as a where
   ...
  AddE :: (a ∈ as)
       ⇒ Exp as a → Exp as a → Exp as a
```

We then seek an instance declaration that allows us to overload the $(+)$ operator in Haskell's Num class, which has type Num $a \Rightarrow a \to a \to a$. The required instance now falls out provided we work the $a \in as$ constraint into the instance header:

```
instance (a ∈ as) ⇒ Num (Exp as a) where
  (+) = AddE
```

Extending this procedure appropriately across the Haskell Prelude's wealth of overloadable functions, we see that it is possible to hide much of the machinery of our technique from the end user. In fact, the only evidence will be the need to supply an instantiation for each type-level list accumulated. For example, let us take our example term, eMixed, and make it more palatable:

```
eMixed
  = if 3 ≡Exp 4 then 0.0 else 3.9
```

Here we have used GHC's support for rebindable syntax to overload Haskell's **if-then-else** expression. The literals 3 and 4 are translated into the applications fromInteger 3 and fromInteger 4 respectively, where fromInteger is a member of the Num class described earlier. In a similar fashion, the floating-point literals are rewritten as calls to the overloaded fromRational function. Haskell's equality operator, $(\equiv)$, is overloadable also, but perhaps ironically is one of a set of operators which do not work nicely with many DSLs. In our case, for example, we would like to implement $(\equiv)$ using the EqE constructor, which has type:

```
(Eq a, a ∈ as) ⇒ Exp as a → Exp as a → Exp as Bool
```

Unfortunately, $(\equiv)$'s type, while overloaded in its arguments (through the Eq class), must always produce a Bool:

```
Eq a ⇒ a → a → Bool
```

We have therefore defined and used an alternative operator, $(\equiv_{\mathsf{Exp}})$, which provides the parametricity we need. For additional convenience we have also specialised $(\equiv_{\mathsf{Exp}})$'s type so that it compares integer expressions only: if its type were to be made more general then we would require type annotations. This is not a problem specific to our technique, however: it is a consequence of the fact that the argument types of EqE (which implements the $(\equiv_{\mathsf{Exp}})$ function) are existential (i.e. they do not appear in the type of its result). Consequently they must be fixed before the type checker is willing to discard the knowledge of their existence. There are alternative solutions to such problems which are common in DSL design, but these fall outside the scope of this paper.

8. The type restriction problem revisited

As we hinted in Section 1 the problem we began with can be solved for the case of rank-1 types using a scheme analogous to that presented by Hughes [11]. The purpose of this section is to outline that solution and to show how it breaks down in the presence of higher-rank types. The idea is to abstract not only over the result type of an expression, but also the target platform itself. By 'target platform' we really mean some abstraction of a specific use of the DSL, although the word 'platform' is particularly appropriate when we think of compiling DSLs.

We first capture the notion that a type a is 'supported by', or is 'typeable on', a particular platform using a type class:

```
class Typeable p a
  valueP :: a → p a
```

We can now use this to specify the constraints on DSL expressions:

```
data Exp p a where
  ValueE :: Typeable p a ⇒ a → Exp p a
  CondE  :: Exp p Bool → Exp p a → Exp p a → Exp p a
  EqE    :: Eq a ⇒ Exp p a → Exp p a → Exp p Bool
```

Here, Exp has been parameterised by a type p which represents the target platform. The ValueE constructor can only be used to lift values whose types are statically known to be representable in the domain of the type p (through the valueP function). Note that while we have once again constrained ValueE's type, the constraint does not tie expressions down to a specific target platform. As an example, compileSM might now be implemented as follows:

```
newtype SM a = SM {fromSM :: Int}
instance IntBool a ⇒ Typeable SM a where
  valueP = SM · toInt
compileSM :: Exp SM a → String
compileSM (ValueE x)
  = "PUSH " ++ show (fromSM (valueP x)) ++ "\n"
```

with the other two cases unchanged. In the ValueE clause, Typeable has taken the role of the IntBool class above whilst valueP is an abstraction of toInt.

So far so good! But let's now build a second function over the DSL in a similar way; here a pretty-printer:

```
newtype Pretty a = Pretty {fromPretty :: String}
```

```
instance Show a ⇒ Typeable Pretty a where
  valueP = Pretty · show

pretty :: Exp Pretty a → String
pretty (ValueE x)
  = fromPretty (valueP x)
pretty (EqE e1 e2)
  = "(" ++ pretty e1 ++ " == " ++ pretty e2 ++ ")"
pretty (CondE p t f)
  = "(if " ++ pretty p ++ " then " ++ pretty t ++
      " else " ++ pretty f ++ ")"
```

This is also perfectly valid and, what's more, it doesn't interfere with compileSM in the sense that neither has restricted the uses of the DSL. However, now suppose that we wish to apply the two functions to the *same* expression, as in:

```
f :: Exp p a → ...
f e = ... (compileSM e) ... (pretty e) ...
```

Type checking the applications of compileSM and pretty will result in the instantiation of p to two different types, namely SM and Pretty. f will therefore fail to type check. This is exactly the problem we saw in Section 5.1 and is a direct result of representing the target platform in an expression's type.

8.1 Hidden types

As we have seen earlier, the usual solution is to introduce a rank-2 type. Let's first try to attach a quantifier to the expression data type:

```
newtype AnyExp a = AnyExp (∀p. Exp p a)
```

f may now be rewritten to accept a value of type AnyExp a as its argument:

```
f :: AnyExp a → ...
f (AnyExp e) = ... (compileSM e) ... (pretty e) ...
```

Does this solve the problem? No! Let's see why:

```
AnyExp (ValueE False)
```

The above expression will not type check due to a missing Typeable p Bool constraint in the type of AnyExp. Of course, we can fix this by altering AnyExp's type:

```
newtype AnyExp a
  = AnyExp (∀p. Typeable p a ⇒ Exp p a)
```

but this will only get us so far. Let's now consider an application of AnyExp to the eMixed expression given in Section 2, which harbours more than one type of ValueE application:

```
AnyExp $
  CondE
    (EqE (ValueE (3 :: Int)) (ValueE (4 :: Int)))
      (ValueE (0.0 :: Float))
      (ValueE (3.9 :: Float))
```

AnyExp's type provides us with the constraint Typeable p Float, but we also need Typeable p Int. We now have a new problem: the constraint Typeable p a can only be verified with respect to the *top-level* expression type; what the type checker needs is a guarantee that *every* type in the expression is typeable on p. To capture this we need to add a constraint to AnyExp for *every* type that we ever expect to encounter in our DSL, thus:

```
newtype AnyExp a
  = AnyExp (∀p. (Typeable p τ1,
                 Typeable p τ2,
                 ...
                 Typeable p τn) ⇒ Exp p a)
```

which is a mess! Worse still, *every* platform must now support *at least* the types $\tau_1, \ldots, \tau_n$ which defeats the purpose of abstracting p in the first place!

Suppose instead, we try to constrain the type of each function over the DSL instead, as in:

```
f :: (∀p. Exp p a) → ...
```

We encounter exactly the same problem as above, but now the constraints are at least localised to the function:

```
f :: (∀p. (Typeable p σ1,
           Typeable p σ2,
           ...
           Typeable p σm) ⇒ Exp p a) → ...
```

where now $\sigma_1, \ldots, \sigma_m$ is an enumeration of all the types common to the instances of p that f requires for its definition (SM and Pretty in the example above). Once again the need to enumerate all these constraints defeats the purpose of abstracting over the platform.

9. Related work

The separation of interface from implementation is a well understood problem. Dynamically typed languages such as Ruby and Python are very flexible in this respect (particularly with regard to DSL embedding) but offer none of the static guarantees we seek in this paper. In contrast, Scala [18] is a statically typed language designed with DSL embedding in mind. Hofer et al. present a technique for embedding DSLs [10] in Scala which is not unrelated to the tagless encodings [5, 21] discussed earlier. Their scheme does not consider the potential consequences of 'impedance mismatches' between multiple implementations, however.

In the functional programming community, Haskell has shown to be a popular host language for a wide variety of embedded DSLs, many of which have been designed to exploit parallelism [1, 2, 7, 15]. Nikola [15] only supports values of type Float so the issues addressed by this paper do not arise. Accelerate [7] is an example of a more powerful language that has been designed with multiple targets in mind but where the designers have opted to constrain the DSL at the point of definition to identify explicitly a single set of supported types. A number of unpublished articles and mailing list discussions raise the issue of data type restriction as defined in this paper: a few offer solutions similar to the one outlined in Section 8. [2]

Phantom types [14] are an alternative tool which may be used for embedding DSLs in which the underlying representation of the DSL is essentially untyped [8]. This offers the designer flexibility when writing functions such as compileSM, but means that a carefully chosen type-safe interface must be exported to the end-user. In our approach we have chosen not to make such a compromise, though the lists of types we use are themselves phantom types, having no impact on a value's representation. Kiselyov's work on implicit configurations [12] also makes use of phantom types and could be applied in the context of constraining data types, though this has not been investigated. Pantheon [25] and Eden [9] are examples of how Template Haskell [26] may be used effectively to manipulate DSLs at compile-time. While this paper has not focused on compile-time techniques, we discuss some potential opportunities in this domain in Section 10.

Dependently typed systems, such as those exhibited by Agda [17] and Idris [3] have already been shown to be useful in DSL construction [4, 20]. They, and the work on SHE [16] have inspired many of the newly-added Haskell features exploited in this paper. However, as we have already pointed out, we only leverage these features

[2] `http://www.haskell.org/pipermail/haskell-cafe/2011-November/096699.html` discusses a technique which uses constraint kinds, for example.

without depending upon them. HList [13] is a Haskell library for working with type-level lists and provides a far greater range of operations and features than those described here.

10. Future work

10.1 Automatic constraint generation

As hinted at in Section 2, we could add generic constraints to the constructors of a type (or equally, the class methods of a tagless representation) mechanically in many cases. In the case of annotating a GADT, $\mathcal{T}$, of kind $\star \to \star$, for example, a naïve algorithm for doing this might proceed as follows:

- Parameterise all occurrences of $\mathcal{T}$ by a list of types as, giving it the new kind $[\star] \to \star \to \star$.
- For each of $\mathcal{T}$'s constructors, $\mathcal{C} :: \tau_1 \to \cdots \to \tau_n \to \mathcal{T}\ as\ a$, add a constraint $b \in as$ for each τ of the form $\mathcal{T}\ as\ b$.

Template Haskell [26] is an extension to Haskell that provides facilities for compile-time metaprogramming (CTMP), and a tool that could be used to implement the above transformation. More interesting however is the question of whether or not a platform's specification (IntBool and its instances, for example) can be generated from a description of the platform at compile-time. Considering our CUDA example from Section 6, it might be possible to infer that the use of conditional operations should be prevented given some first-class representation of the fact that it is a single instruction, multiple thread (SIMT) architecture.

10.2 Eliminating type annotations

Somewhat frustrating is the need to supply type annotations whenever a list of types is eliminated, viz.:

```
compileSM (ValueE 3 :: Exp '[Int] Int)
```

In the case of compileSM, this burden may be removed by creating a compiler with a more grounded type signature:

```
compileSM' :: Exp '[Int, Bool] a → String
compileSM' = compileSM
```

Such a definition permits the expression compileSM' (ValueE 3) while others such as compileSM' eMixed are rejected as ill-typed. This technique will not suffice for all targets however. Consider a Haskell evaluator for our GADT expressions:

```
evaluate :: Exp as a → a
evaluate (ValueE x)     = x
evaluate (EqE e1 e2)    = evaluate e1 ≡ evaluate e2
evaluate (CondE p t f)  = if evaluate p
                          then evaluate t
                          else  evaluate f
```

In this case there are an infinite number of types we could instantiate as to. This is particularly annoying when, given a principal type, there is a mechanical translation from its list of $(\cdot \in \cdot)$ constraints to a minimal satisfying list of types. For example, the type:

$$(\mathsf{Int} \in as, \mathsf{Bool} \in as, \mathsf{Float} \in as) \Rightarrow \mathsf{Exp}\ as\ \mathsf{Float}$$

may always be instantiated to:

$$\mathsf{Exp}\ '[\mathsf{Int}, \mathsf{Bool}, \mathsf{Float}]\ \mathsf{Float}$$

Each constraint of the form $a \in as$ results in the type a being added to the list of types we shall pick. We can encode this type-level function directly, using either GADTs or GHC's support for *type families* [6]:

```
type family Satisfying (c :: Constraint) :: [⋆]
type instance Satisfying (a ∈ as)    = '[a]
type instance Satisfying (a ∈ as, c) = a : Satisfying c
```

However, using such a family proves impossible without a type annotation due to the issues involved in 'trapping' type classes mentioned in Section 3 (see the case for Trap). Furthermore, we would then have to convince the compiler that the type Satisfying c really does satisfy the constraint c. We expect that such a process is non-trivial but it merits further investigation.

10.2.1 Making use of metaprogramming

If we are willing to ask for the compiler's help, Template Haskell provides a function reify which allows one to obtain information about an identifier at compile-time. This includes an ADT representation of the identifier's type, from which we can surely implement the translation realised as a type family above. Moreover, since such an instantiation would occur during type checking, the compiler would be able to verify whether or not the types picked satisfy the constraints being eliminated. However, due to some of Template Haskell's practical limitations, such as not being able to reify names defined in the same module (see the work of Sheard and Peyton Jones [26] for more information), we have not yet implemented such functionality. A key point also is that reify can only operate on *named* values. In this respect it would be beneficial to have something similar to C++'s `decltype` operator, which is capable of returning the declared type of an arbitrary expression.

11. Conclusions

We have presented what is essentially a design pattern for the type-safe separation of an interface from multiple implementations. Each implementation may enforce different typing requirements and restrictions on a data type without limiting other uses of the same type. Interestingly, in some situations we are able to support an element of re-use that would ordinarily require the introduction of a rank-2 type, for example when applying multiple functions to the same DSL expression. This turns out to be useful when building DSL compilers for exploiting heterogeneous data parallelism, for example, where we may need to implement a DSL program on more than one target architecture. When there is a need for higher-rank types, for example to support the same type of re-use in a tagless DSL representation, our method applies equally well. The idea of imposing only generic, rather than concrete, constraints on a data type thus has a number of advantages.

Somewhat as an aside, we have also found many of Haskell's latest type features, in particular first-class constraints and type promotion, to be very useful, although, as we have noted, it is possible to achieve the same effect without them.

References

[1] L. Augustsson, H. Mansell, and G. Sittampalam. Paradise: A Two-Stage DSL Embedded in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 225–228, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7.

[2] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The Design and Implementation of Feldspar - An Embedded Language for Digital Signal Processing. In *Proceedings of the 22nd Symposium on Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 121–136. Springer-Verlag, 2010. ISBN 978-3-642-24275-5.

[3] E. C. Brady. IDRIS – Systems Programming Meets Full Dependent Types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*, PLPV '11, pages 43–54, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0487-0.

[4] E. C. Brady and K. Hammond. Scrapping your Inefficient Engine: Using Partial Evaluation to Improve Domain-Specific Language Implementation. *ACM SIGPLAN Notices*, 45(9):297–308, September 2010. ISSN 0362-1340.

[5] J. Carette, O. Kiselyov, and C. chieh Shan. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming*, 19:509–543, September 2009. ISSN 0956-7968.

[6] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated Types with Class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 1–13, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X.

[7] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0486-3.

[8] C. Elliott, S. Finne, and O. de Moor. Compiling Embedded Languages. *Journal of Functional Programming*, 13(3):455–481, 2003. ISSN 0956-7968.

[9] K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, September 2003. ISSN 0129-6264.

[10] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic Embedding of DSLs. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 137–148, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2.

[11] J. Hughes. Restricted Data Types in Haskell. In *Proceedings of the 1999 Haskell Workshop*. University of Utrecht, Technical Report UU-CS-1999-28, October 1999.

[12] O. Kiselyov and C. chieh Shan. Functional Pearl: Implicit Configurations–or, Type Classes Reflect the Values of Types. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 33–44, New York, NY, USA, 2004. ACM. ISBN 1-58113-850-4.

[13] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107, New York, NY, USA, 2004. ACM. ISBN 1-58113-850-4.

[14] D. Leijen and E. Meijer. Domain-Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages - Volume 2*, DSL'99, pages 109–122, Berkeley, CA, USA, 1999. USENIX Association.

[15] G. Mainland and G. Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the 3rd ACM Haskell Symposium*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4.

[16] C. McBride. Faking It: Simulating Dependent Types in Haskell. *Journal of Functional Programming*, 12:375–392, July 2002. ISSN 0956-7968.

[17] U. Norell. *Towards a Practical Programming Language based on Dependent Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[18] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[19] D. A. Orchard and T. Schrijvers. Haskell Type Constraints Unleashed. In *Proceedings of the 10th International Symposium on Functional and Logic Programming*, Lecture Notes in Computer Science, pages 56–71. Springer-Verlag, 2010. ISBN 978-3-642-12251-4.

[20] N. Oury and W. Swierstra. The Power of Pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7.

[21] E. Pašalić, W. Taha, and T. Sheard. Tagless Staged Interpreters for Typed Languages. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 218–229, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8.

[22] S. Peyton Jones, M. Jones, and E. Meijer. Type Classes: Exploring the Design Space. In *Proceedings of the 1997 Haskell Workshop*, 1997.

[23] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical Type Inference for Arbitrary-Rank Types. *Journal of Functional Programming*, 17:1–82, January 2007. ISSN 0956-7968.

[24] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and Decidable Type Inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 341–352, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.

[25] S. Seefried, M. M. T. Chakravarty, and G. Keller. Optimising Embedded DSLs Using Template Haskell. In *GPCE '04*, volume 3286 of *Lecture Notes in Computer Science*, pages 186–205. Springer-Verlag, 2004. ISBN 3-540-23580-9.

[26] T. Sheard and S. Peyton Jones. Template Meta-Programming for Haskell. *SIGPLAN Notices*, 37:60–75, December 2002. ISSN 0362-1340.

[27] D. Vytiniotis, S. Weirich, and S. Peyton Jones. Boxy Types: Inference for Higher-Rank Types and Impredicativity. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 251–262, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3.

[28] P. Wadler and S. Blott. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2.

[29] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. M. aes. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5.