

Ad-hoc Polymorphism and Dynamic Typing in a Statically Typed Functional Language

Thomas van Noort Peter Achten Rinus Plasmeijer

Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL, Nijmegen, The Netherlands

{thomas, p.achten, rinus}@cs.ru.nl

Abstract

Static typing in functional programming languages such as Clean, Haskell, and ML is highly beneficial: it prevents erroneous behaviour at run time and provides opportunities for optimisations. However, dynamic typing is just as important as sometimes types are not known until run time. Examples are exchanging values between applications by deserialisation from disk, input provided by a user, or obtaining values via a network connection. Ideally, a static typing system works in close harmony with an orthogonal dynamic typing system; not discriminating between statically and dynamically typed values. In contrast to Haskell's minimal support for dynamic typing, Clean has an extensive dynamic typing; it adopted ML's support for monomorphism and parametric polymorphism and added the notion of type dependencies. Unfortunately, ad-hoc polymorphism has been left out of the equation over the years. While both ad-hoc polymorphism and dynamic typing have been studied in-depth earlier, their interaction in a statically typed functional language has not been studied before. In this paper we explore the design space of their interactions.

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

General Terms Design, Languages

Keywords ad-hoc polymorphism, dynamic typing

1. Introduction

Static typing is the cornerstone of functional programming languages such as Clean, Haskell, and ML. It prevents erroneous behaviour at run time by verifying type safety at compile time. Also, it provides opportunities for optimisations by exploiting either user-specified or inferred type information statically.

However, sometimes the type of a value is not known until run time. Typically this is the case when interacting with the 'outside' world: exchanging values between applications by deserialisation from disk, input provided by a user, or obtaining values via a network connection. In such cases, dynamic typing is required to defer type unification until run time. Values are wrapped in a uniform black box, as their type is statically not known, and

unwrapped by pattern matching and providing a required type. Although type unification can fail at run time when a dynamic value presents an unexpected type, the static type system guarantees that when pattern matching succeeds, the unwrapped value can be used safely.

While many dispute over choosing either static or dynamic typing, we agree that the solution lies in the middle (Meijer and Drayton, 2004):

“Static typing where possible, dynamic typing when needed”

We believe that a statically typed language is the starting point, extended with an escape to type values dynamically. Ideally, the dynamic typing system is orthogonal to the static typing system, imposing no restrictions on the value or types that can be considered dynamic.

Haskell has minimal support for dynamic typing, it only supports monomorphism (Baars and Swierstra, 2002; Cheney and Hinze, 2002). Clean, on the other hand, has a rich and mature dynamic typing system; it adopted ML's support for monomorphism (Abadi et al., 1991; Pil, 1997) and parametric polymorphism (Abadi et al., 1994; Leroy and Mauny, 1993). Additionally, it includes the notion of type dependencies (Pil, 1999). Even generic functions can be applied to dynamic values (Wichers Schreur and Plasmeijer, 2005). Though, the quest for an orthogonal dynamic typing system cannot be completed without proper support for another important concept: ad-hoc polymorphism.

Ad-hoc polymorphism provides an abstraction mechanism to parameterise values with behaviour. The usual suspects are functions for the equality and ordering of values. Whereas in ML ad-hoc polymorphism is modelled via the module system (Wehr and Chakravarty, 2008), Haskell and Clean model ad-hoc polymorphism via type classes which is resolved to a dictionary-passing style at compile time (Peterson and Jones, 1993; Wadler and Blott, 1989). Static type information is crucial in this approach; it is the driving force behind the translation mechanisms.

Although both ad-hoc polymorphism and dynamic typing have been studied in-depth earlier, their interaction in a statically typed functional language has not been explored before. We identify two sides to their interaction. On the one hand, it involves dynamic typing in the world of ad-hoc polymorphism. For instance, a sorting function applied to a list of dynamically typed values that are obtained by deserialisation from disk or provided by a user as input. Obviously, this poses a challenge since ad-hoc polymorphism is resolved at compile time while the type of dynamic values is known only at run time. Typically, this is solved by enumerating all expected types by hand. This resolves ad-hoc polymorphism statically but is cumbersome, prone to errors, and does not scale for evident

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WGP'10, September 26, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0251-7/10/09...\$10.00

reasons. On the other hand, the interaction concerns ad-hoc polymorphism in the world of dynamic typing. For example, a sorting function that is deserialised from disk and applied to some statically typed value. Here, the challenge is to extend the existing dynamic typing mechanisms to support ad-hoc polymorphic values.

In this paper we explore the design space of the interactions and provide a thorough intuition of the issues involved. While there is a plethora of type class extensions (Peyton Jones et al., 1997), we first only consider type classes in the style of Haskell 98 (Peyton Jones, 2003). We set the scene by giving an overview of both conventional ad-hoc polymorphism via type classes in Clean and Haskell (Section 2), and dynamic typing in Clean (Section 3). Our contributions are the following:

- We describe two complementary approaches to dynamic typing in ad-hoc polymorphism (Section 4): container datatypes and dynamic dictionary composition.
- We describe two different approaches to ad-hoc polymorphism in dynamic typing (Section 5): dictionary-passing types and type code extension.
- We discuss how several type class extensions affect both sides of the interaction (Section 6).

Finally, we elaborate on related work (Section 7) and conclude with a brief discussion and future work (Section 8).

The examples given in this paper are defined using Clean syntax. While types in Clean have an explicit arity, we curry the types for the sake of presentation. An overview of syntactical and semantical differences between Clean and Haskell is given elsewhere (Achten, 2007; Van Groningen et al., 2010).

2. Ad-hoc polymorphism

We give a brief overview of type classes (Section 2.1) and their translation to dictionary-passing style (Section 2.2).

2.1 Type classes

Consider the following type class for the equality of values:

```
class Eq α where
  eq :: α → α → Bool
```

The type class *Eq* has one member, the equality function. We ignore default members for simplicity reasons; these are irrelevant to our approach and only clutter the examples. We provide several instances for this type class:

```
instance Eq Int where
  eq x y = eqInt x y
```

```
instance Eq [α] | Eq α where
  eq x y = eq (length x) (length y) ∧ and (zipWith eq x y)
```

```
instance Eq (α, β) | Eq α & Eq β where
  eq x y = eq (fst x) (fst y) ∧ eq (snd x) (snd y)
```

We assume that in the instance for integers, a core equality function *eqInt* is available. In the instance for lists, we require there to be an instance for the element type as well since we pairwise compare the elements. Similarly in the instance for pairs, we require instances for both element types.

Next, we define a type class for the ordering of values:

```
class Ord α | Eq α where
  lt :: α → α → Bool
```

This type class has one member function as well, one that tests if the first argument is ‘less than’ the second argument. We also see that the *Eq* type class is a superclass of the *Ord* class, denoting

```
:: DictEq α = { eq :: α → α → Bool }
dictEqInt :: DictEq Int
dictEqInt = { eq = λx y → eqInt x y }
dictEqList :: ∀ α . DictEq α → DictEq [α]
dictEqList da =
  { eq = λx y → dictEqInt.eq (length x) (length y)
    ∧ and (zipWith da.eq x y) }
dictEqPair ::
  ∀ α β . DictEq α → DictEq β → DictEq (α, β)
dictEqPair da db =
  { eq = λx y → da.eq (fst x) (fst y)
    ∧ db.eq (snd x) (snd y) }

:: DictOrd α = { lt :: α → α → Bool
               , dictEq :: DictEq α }
dictOrdInt :: DictOrd Int
dictOrdInt = { lt = λx y → ltInt x y
              , dictEq = dictEqInt }
dictOrdList :: ∀ α . DictOrd α → DictOrd [α]
dictOrdList da =
  { lt = λx y → dictOrdInt.lt (length x) (length y)
    ∨ or (zipWith da.lt x y)
    , dictEq = dictEqList da.dictEq }
dictOrdPair ::
  ∀ α β . DictOrd α → DictOrd β → DictOrd (α, β)
dictOrdPair da db =
  { lt = λx y → da.lt (fst x) (fst y)
    ∨ db.lt (snd x) (snd y)
    , dictEq = dictEqPair da.dictEq db.dictEq }
```

Figure 1. Dictionary-passing style translation of *Eq* and *Ord*

that every *Ord* instance is also an *Eq* instance. Again, we define instances for integers, lists, and pairs:

```
instance Ord Int where
  lt x y = ltInt x y
```

```
instance Ord [α] | Ord α where
  lt x y = lt (length x) (length y) ∨ or (zipWith lt x y)
```

```
instance Ord (α, β) | Ord α & Ord β where
  lt x y = lt (fst x) (fst y) ∨ lt (snd x) (snd y)
```

We assume the presence of a core function *ltInt* for the ordering of integers. Similar to the instance for *Eq*, the instances for lists and pairs require instances for their element types. Admittedly, not all of these instances are useful in practice. Here, they merely serve the purpose of illustrating the dictionary-passing style translation.

A typical use of the ordering type class is a sorting function:

```
sort :: ∀ α . [α] → [α] | Ord α
```

For the sake of brevity we leave its definition abstract. The type of *sort* reflects that it sorts a list of values, constrained by a context *Ord* of the element type. Note that since *Eq* is a superclass of *Ord*, this makes the *lt* function as well as the *eq* function available to the sorting function.

2.2 Dictionary-passing style

Type classes are translated at compile time to a dictionary-passing style. Each type class definition translates to a dictionary type that captures its members and superclasses. Then, each instance is an

instantiation of that dictionary type. For example, in Figure 1 we see the dictionary-passing style translation of the *Eq* and *Ord* type class and their instances. The *Eq* type class translates to a record type *DictEq* that has one field for its member function *eq*. The *DictOrd* record has a field for its member *lt* and an additional field for its super class *Eq* dictionary. Each body of the instances is visible in the instantiations of the dictionary types. The instances that require instances for their element types, such as for lists and pairs, are passed additional dictionaries. In *dictEqList* and *dictOrdList* we also see how a concrete dictionary is used to test the length of the argument lists.

Then, ad-hoc polymorphic values are translated such that they receive additional dictionary arguments. For example, the dictionary-passing type of the sorting function becomes:

$$\text{sort} :: \forall \alpha . \text{DictOrd } \alpha \rightarrow [\alpha] \rightarrow [\alpha]$$

Any occurrence of the sorting function inserts the appropriate dictionary. For instance, consider the following expression:

$$\begin{aligned} &\text{let } x = [1..10] \\ &\text{in } eq \\ &\quad (\text{sort } x) \ x \end{aligned}$$

Resolving the occurrences of *eq* and *sort* results in the following expression in dictionary-passing style:

$$\begin{aligned} &\text{let } x = [1..10] \\ &\text{in } (\text{dictEqList } \text{dictEqInt}).eq \\ &\quad (\text{sort } (\text{dictOrdList } \text{dictOrdInt}) \ x) \ x \end{aligned}$$

Here we see that the equality function is replaced by accessing the appropriate record in the equality dictionary for a list of integers. The sorting function is provided an additional argument, namely, the ordering dictionary for a list of integers.

3. Dynamic typing

Next, we provide a crash course (not to be taken literally) in Clean's dynamic typing system, discussing monomorphism (Section 3.1), parametric polymorphism (Section 3.2), type dependencies (Section 3.3), and type codes (Section 3.4).

3.1 Monomorphism

In Clean, dynamic typing allows monomorphic values to be wrapped together with their type in a uniform package. This is called a dynamic, and is obtained using the corresponding keyword, thereby obtaining a value of the type *Dynamic*:

$$\begin{aligned} \text{wrapInt} &:: \text{Int} \rightarrow \text{Dynamic} \\ \text{wrapInt } x &= \mathbf{dynamic} \ x :: \text{Int} \end{aligned}$$

Using the $::$ annotation, we explicate the type of the value that is wrapped. The annotation is optional and only required when the type cannot be inferred.

A dynamic value is unwrapped by pattern matching and providing a required type using the $::$ annotation, for example to obtain an integer from a dynamic:

$$\begin{aligned} \text{unwrapInt} &:: \text{Dynamic} \rightarrow \text{Int} \\ \text{unwrapInt } (x :: \text{Int}) &= x \\ \text{unwrapInt } (x :: \text{String}) &= \text{stringToInt } x \\ \text{unwrapInt } _ &= \perp \end{aligned}$$

The first arm pattern matches on integer values, returning the value itself if that is the case. If the value in the dynamic is a string, we convert it to an integer. As type unification takes place at run time and pattern matching can fail, a catch-all arm is required for totality; either returning a default value or a run-time error message. For the sake of convenience, we choose to return \perp for failed dynamic pattern matches in this paper.

Instead of enumerating every possible type, pattern variables can be used in the type of a dynamic pattern match. Typically, this is used to enforce type equality between dynamic values, for instance in the infamous example of dynamic function application:

$$\begin{aligned} \text{dynApp} &:: \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Dynamic} \\ \text{dynApp } (f :: a \rightarrow b) \ (x :: a) &= \mathbf{dynamic} \ (f \ x) \\ \text{dynApp } _ \ _ &= \perp \end{aligned}$$

Pattern variables, denoted here by roman instead of greek characters, in a single arm definition share the same scope. Therefore, the first arm only succeeds once the argument type of the function matches the type of the argument. Then, the result is wrapped in a dynamic again.

3.2 Parametric polymorphism

Besides monomorphic values, parametric polymorphic values can be (un)wrapped as well without any additional effort. For example, a function that does not change the type of its argument is wrapped as follows:

$$\begin{aligned} \text{wrapFun} &:: (\forall \alpha . \alpha \rightarrow \alpha) \rightarrow \text{Dynamic} \\ \text{wrapFun } f &= \mathbf{dynamic} \ f \end{aligned}$$

Analogously to unwrapping integers, a parametric polymorphic function is unwrapped by specifying the required type in a dynamic pattern match:

$$\begin{aligned} \text{unwrapFun} &:: \text{Dynamic} \rightarrow (\forall \alpha . \alpha \rightarrow \alpha) \\ \text{unwrapFun } (f :: \forall \alpha . \alpha \rightarrow \alpha) &= f \\ \text{unwrapFun } _ &= \perp \end{aligned}$$

The α as occurring in the type of the function is different from the same type variable in the dynamic pattern match; both have different binding sites.

Note that dynamic pattern matches can contain both type variables (α , β , etc.) as well as pattern variables (a , b , etc.), the difference being that the former are explicitly bound by a universal quantifier while the latter are not. Consider the following function that tries to unwrap a function and apply it to a list:

$$\begin{aligned} \text{dynAppList} &:: \forall \alpha . \text{Dynamic} \rightarrow [\alpha] \rightarrow \text{Dynamic} \\ \text{dynAppList } (f :: \forall \alpha . [\alpha] \rightarrow b) \ x &= \mathbf{dynamic} \ (f \ x) \\ \text{dynAppList } _ \ _ &= \perp \end{aligned}$$

Here, the first arm only succeeds if the dynamic contains a function that transforms any list regardless of the type of its elements, such as *length* $:: \forall \alpha . [\alpha] \rightarrow \text{Int}$, *head* $:: \forall \alpha . [\alpha] \rightarrow \alpha$, but also concatenation using the operator ($++$) $:: \forall \alpha . [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$.

3.3 Type dependencies

The previous examples are context independent, in other words, the process of (un)wrapping values is not determined by the context in which these functions are used. Type dependencies allow the context to guide the (un)wrapping of values. A straightforward example is the following function that wraps any value in a dynamic:

$$\begin{aligned} \text{wrap} &:: \forall \alpha . \alpha \rightarrow \text{Dynamic} \mid \text{TC } \alpha \\ \text{wrap } x &= \mathbf{dynamic} \ x \end{aligned}$$

Here, the function is (ad-hoc) polymorphic in the argument type. We require the context to provide a so-called type code (i.e., the value representation of the type) using Clean's built-in type class *TC*, which is stored together with the value. We elaborate later in Section 3.4 on this type class and type codes.

Similarly, we can unwrap values depending on the context:

$$\begin{aligned} \text{unwrap} &:: \forall \alpha . \text{Dynamic} \rightarrow \alpha \mid \text{TC } \alpha \\ \text{unwrap } (x :: \alpha^\wedge) &= x \\ \text{unwrap } _ &= \perp \end{aligned}$$

We require the value to be of the function result type by referring to the binding site of the same type variable using the \wedge annotation, omitting the universal quantifier from the dynamic pattern match. This causes the type code from the dynamic to be unified with the type code obtained from the context, denoted by the TC context. Therefore, success of the pattern match depends on the context in which the value is unwrapped.

As another example, we redefine $dynApp$ from Section 3.1 such that it depends on the context:

$$\begin{aligned} dynApp &:: \forall \beta . Dynamic \rightarrow Dynamic \rightarrow Maybe \beta \mid TC \beta \\ dynApp (f :: a \rightarrow \beta^\wedge) (x :: a) &= f x \\ dynApp _ _ &= \perp \end{aligned}$$

The function now only succeeds if the return type of the first argument fits the context, denoted by the type variable β and the use of the \wedge annotation. Again, the TC context is required so that the type code from the context can be compared to the type code stored in the dynamic.

3.4 Type codes

As alluded to in the previous sections, type codes lie at the heart of dynamic typing; whenever a value is wrapped in a dynamic, a type code is included as well. Also, functions with type dependencies require a type code to unify type information from the context with type information stored in a dynamic. In Clean, a type code is obtained via the built-in type class TC :

```
class TC  $\alpha$  where
  typeCode :: TypeCode
```

It has a single member constant that provides a type code for its type argument. However, this type class is treated specially: any instance that is required is generated at compile time. Therefore, we cannot provide manual instances of this type class. Type codes are defined by the vanilla datatype $TypeCode$:

```
:: TypeCode = Scheme [String] TypeCode
             | Con TypeDef
             | App TypeCode TypeCode
             | Var String
:: TypeDef
```

A type code represents a universally quantified type with a list of variables, which is typically empty for monomorphic types. A type constructor is represented by a type definition, whose definition is left abstract for the sake of presentation. The definition of a type must be included since dynamics can be (de)serialised across application boundaries. Then, verifying name equivalence during type unification simply does not suffice. Consequently, type codes cannot be defined for abstract types, but only for any nonabstract type. The other alternatives of $TypeCode$ represent type application and type variables. As an example, the type $\forall \alpha . [\alpha] \rightarrow Int$ of the function $length$ is represented by:

```
Scheme ["a"]
  (App (App (Con funDef)
            (App (Con listDef)
                  (Var "a"))))
      (Con intDef))
```

The definition of the function, list, and integer type is left abstract.

4. Dynamic typing in ad-hoc polymorphism

After the overview of both participants, we continue by considering the first side of their interaction: dynamic typing in the world of ad-hoc polymorphism. As a running example in this section, we consider applying the sorting function from Section 2.1 to a list

that is unwrapped from a dynamic value. Naively, such a function would be defined as follows:

$$\begin{aligned} dynSort &:: Dynamic \rightarrow Dynamic \\ dynSort (x :: [a]) &= \mathbf{dynamic} (sort x) \\ dynSort _ &= \perp \end{aligned}$$

Since ad-hoc polymorphism is resolved at compile time, the challenge here is that critical type information becomes apparent only at run time. Consequently, it is not known at compile time which instance must be provided to the sorting function. Alternatively, we could define a similar function that exposes the result type using type dependencies and the \wedge annotation from Section 3.3:

$$\begin{aligned} dynSort &:: \forall \alpha . Dynamic \rightarrow [\alpha] \mid TC \alpha \\ dynSort (x :: [\alpha^\wedge]) &= sort x \\ dynSort _ &= \perp \end{aligned}$$

Here, the dynamic type of the elements in the list is related to a static type. Though, ad-hoc polymorphism still cannot be resolved. The type code is provided as an abstract argument to this function, which cannot be used for resolving purposes at compile time.

A straightforward solution would be to define this function without a pattern variable and enumerate all expected types:

$$\begin{aligned} dynSort &:: Dynamic \rightarrow Dynamic \\ dynSort (x :: [Int]) &= \mathbf{dynamic} (sort x) \\ dynSort (x :: [[Int]]) &= \mathbf{dynamic} (sort x) \\ dynSort (x :: [[[Int]]]) &= \mathbf{dynamic} (sort x) \\ \dots & \\ dynSort _ &= \perp \end{aligned}$$

This would resolve ad-hoc polymorphism at compile time since all required type information is provided manually. Evidently, this approach is cumbersome, prone to errors, and does not scale: we have to duplicate the right-hand side of the original function definition and easily forget an arm. Moreover, there are often an infinite number of alternatives.

In this section, we specifically consider the situations where resolving ad-hoc polymorphism at compile time relies on type information that only becomes available at run time via pattern variables. We describe two complementary approaches to this challenge: container types (Section 4.1) and dynamic dictionary composition (Section 4.2).

4.1 Container datatypes

The first approach makes the producer of a dynamic value responsible for resolving ad-hoc polymorphism in future uses of this value. This is modelled by datatypes containing both values and their available instances, dubbed container datatypes; similar to classes in object-oriented programming languages. The most well-known form of container datatypes is existential datatypes (Läufer and Odersky, 1994). For example, the following datatype encapsulates a list value that can be ordered by the type of its elements:

$$:: EContOrdList = \exists \alpha . EContOrdList ([\alpha] \mid Ord \alpha)$$

The container prohibits the type of its value from escaping, and only permits ordering operations. Since the notation for datatype alternatives coincides with Clean's notation for type class contexts, we explicitly provide parentheses to denote that there must be an instance available for the type of the value, instead of a second alternative for the $ContOrdList$ type. A more permissive form of container datatypes is the following:

$$:: ContOrdList \alpha = ContOrdList ([\alpha] \mid Ord \alpha)$$

The existential type is pushed out of the definition such that the type of the value is exposed by the container. We often need such exposure to apply other operations than only the ones permitted or

to relate dynamics to each other using pattern variables. Be aware that the semantics of such container datatypes is different from Haskell’s analogues definition:

```
data Ord α ⇒ ContOrdList α = ContOrdList [α]
```

Here, the context only guarantees that an instance exists, while Clean’s approach also makes the corresponding instance available when the constructor is pattern matched. Typically, the latter behaviour is achieved in Haskell using generalised algebraic datatypes (Peyton Jones et al., 2006):

```
data ContOrdList :: * → * where
  ContOrdList :: Ord α ⇒ [α] → ContOrdList [α]
```

We use Clean’s container datatypes because generalised algebraic datatypes are not yet supported and we do not need its full power to model container datatypes.

4.1.1 Semantics

We define *dynSort* again; now using the container datatype for *Ord* on lists:

```
dynSort :: Dynamic → Dynamic
dynSort (ContOrdList x :: ContOrdList a) =
  dynamic (sort x)
dynSort _ = ⊥
```

The dynamic pattern match is changed to include the (type) constructor of the container datatype. Note that we cannot use the existential variant here. Otherwise, its hidden element type would escape to the type code included in the resulting dynamic value.

The context that is introduced by the use of *sort* is statically fulfilled by the local context that is propagated by pattern matching the container datatype. The semantics are very similar to context introduced by existential datatypes and generalised algebraic datatypes (Peyton Jones et al., 2010). Concretely, in dictionary-passing style, the container datatype for *Ord* carries a dictionary in an extra field:

```
:: ContOrdList α = ContOrdList [α] (DictOrd α)
```

Every construction of a container datatype fills in the appropriate dictionary, which is accessed by the dynamic sorting function:

```
dynSort :: Dynamic → Dynamic
dynSort (ContOrdList x da :: ContOrdList a) =
  dynamic (sort da x)
dynSort _ = ⊥
```

The obtained dictionary is passed on to the sorting function.

4.1.2 Discussion

The main advantage of container datatypes is that it is more a static approach than a dynamic approach. It does require additional plumbing through (type) constructors, but imposes a minimal runtime overhead. We specify the context beforehand, allowing the corresponding dictionaries to be inserted at compile time. The downside is that it requires us to predict all required context in advance, something which can be hard. Therefore, this approach is better suited for applications that do not require much flexibility and are confined to strict interfaces. For instance, when values are exchanged between applications and the permitted operations need to be restricted.

A more worrying problem is that ambiguities quickly arise when multiple container datatypes are used. For example, when we define a dynamic equality function. First, we define another container datatype that captures any value with its *Ord* instance:

```
:: ContOrd α = ContOrd (α | Ord α)
```

Then, we define the dynamic equality function as follows:

```
dynEq :: Dynamic → Dynamic → Bool
dynEq (ContOrd x :: ContOrd a)
      (ContOrd y :: ContOrd a) = eq x y
dynEq _ _ = ⊥
```

Here, we statically enforce type equality of the two values by reusing the pattern variable *a*. Also, we require both values to be in a container together with their *Ord* instance. However, there is no guarantee that the instance in the first value is semantically equivalent to the instance in the second value. Possibly, these dynamic values stem from different applications. It is only guaranteed that both values have an instance available. Also, it is unspecified which one to choose. The same issues arise when static contexts are mixed with dynamic contexts:

```
dynEq :: ∀ α . Dynamic → α → Bool | Ord α & TC α
dynEq (ContOrd x :: ContOrd α^) y = eq x y
dynEq _ _ = ⊥
```

Again, it is unclear whether to choose the instance for *Eq* obtained from the container datatype or the context. We believe it is better to refuse such definitions statically than to implement a complicated heuristic that solves the ambiguities arbitrarily. A straightforward manual solution is to remove a container datatype constructor using a helper function or a context from the function type, depending on the desired behaviour.

4.2 Dynamic dictionary composition

In contrast to the first approach, the second approach makes the consumer of a dynamic value responsible for resolving ad-hoc polymorphism. In other words, the function that pattern matches a dynamic value has to come up with the appropriate instance. Since this depends on type information that becomes available at run time, we have to perform this process dynamically. Instead of translating the well-known static mechanism completely to their dynamic counterpart, we keep the quote of Meijer and Drayton from the introduction in mind and only translate those parts across the dynamic border that cannot be performed statically. More concretely, only the composition of dictionary definitions needs to occur at run time.

4.2.1 Semantics

Again, we redefine *dynSort*; now using an explicit context in the dynamic pattern match:

```
dynSort :: Dynamic → Dynamic
dynSort (x :: [a] | Ord a) = dynamic (sort x)
dynSort _ = ⊥
```

The required context introduced by the use of *sort* is now fulfilled by the dynamic pattern match context. The pattern-match semantics of such contexts in a dynamic pattern match are straightforward: when there is no instance available at run time for the matched pattern variable, the pattern match fails. The requirement is in addition to the original dynamic pattern match semantics; the two-stage process is explicated in the translated dictionary-passing style definition of *dynSort*:

```
dynSort :: Dynamic → Dynamic
dynSort (x :: [a])
  | gda = dynamic (sort da x)
where
  (gda, da) = guards (genDictOrd (dynamic ⊥ :: a))
dynSort _ = ⊥
```

The elegance of the translation is that this definition uses conventional dynamic typing mechanisms. In other words, the added con-

text notation is merely syntactic sugar. The context is pushed out of the dynamic pattern match and turned into a guard that verifies the presence of the required instance. The corresponding dictionary is obtained at run time by a generator function, given the type for which it has to compose one. This function groups the available instances and is mechanically deduced at compile time. We provide it the matched pattern variable as a value using a trick: we construct a dynamic value that is \perp , relying on lazy evaluation, and explicitly provide an annotation that this value is of type a . The resulting dictionary for this type is prepared by the helper function *guards* such that we can use the fall-through semantics of the guard:

```
guards :: ∀ α . Maybe Dynamic → (Bool, α) | TC α
guards (Just (x :: α^)) = (True, x)
guards Nothing         = (False, ⊥)
```

The second element of the resulting pair is only used when the first element passes a guard.

There is not necessarily a dictionary available for any type. Also, because the type of each dictionary is different, the result of the generator function is wrapped in a dynamic again. Hence, its type becomes:

```
genDictOrd :: Dynamic → Maybe Dynamic
```

Since this type permits almost any argument and result type, we have to keep correctness by construction in mind. The invariant we are guarding here is that given a value of type α , a dictionary of type *DictOrd* α is returned, if there is one available. Typically, this is expressed through generalised algebraic datatypes, but Clean does not support such definitions as mentioned before in Section 4.1.

Recall from Section 2.1 that there are instances of *Ord* for integers, lists, and pairs. Each arm of the generator function follows mechanically from the available instances, each returning the corresponding dictionary from Figure 1. The first arm follows from the instance for integers:

```
genDictOrd (⊥ :: Int) = Just (dynamic dictOrdInt)
```

The corresponding *Ord* dictionary for integers is simply returned. The arm for lists requires a bit more work, using *unwrap* as defined in Section 3.3:

```
genDictOrd (⊥ :: [a]) =
  do da ← genDictOrd (dynamic ⊥ :: a)
     Just (dynamic (dictOrdList (unwrap da)))
```

As the instance header for lists dictates, a dictionary for *Ord* of the element type is required. Therefore, we match the type using a pattern variable and generate an *Ord* dictionary accordingly, borrowing Haskell's **do** notation for the sake of handling *Maybe* values conveniently. If this succeeds, we unwrap the result and construct the final dictionary for *Ord* of lists. Note that in general, unwrapping a dynamic value can fail. Here, we rely on correctness by construction as mentioned earlier. Similarly, the arm for pairs follows mechanically from its instance:

```
genDictOrd (⊥ :: (a, b)) =
  do da ← genDictOrd (dynamic ⊥ :: a)
     db ← genDictOrd (dynamic ⊥ :: b)
     Just (dynamic (dictOrdPair (unwrap da)
                               (unwrap db)))
```

We bind the different element types to pattern variables and generate *Ord* dictionaries for both element types. Then, if both result in a dictionary, we unwrap the results and generate a dictionary for *Ord* of pairs. Finally, a catch-all arm is defined if the presented type is none of the above (i.e., there is no instance available for this type):

```
genDictOrd ⊥ = Nothing
```

Note that though *Eq* is a superclass of *Ord*, we do not have to consider its dictionary composition since these are already included in the available dictionaries for *Ord*. The generator function merely composes these definitions as dictated by the available instances.

4.2.2 Discussion

Opposite to container datatypes, dynamic dictionary composition is more a dynamic approach; composition takes place at run time using a compile-time deduced generator function. Consequently, this approach is likely to introduce more overhead at both compile time and run time. On the other hand, we are not confronted with the additional plumbing of (type) constructors of container datatypes. But above all, we do not have to know the required contexts in advance; operations are separated from values. Therefore, this approach is more suited to applications that require more flexibility where less assumptions can be made about the purpose of dynamically typed values. For example, when values are obtained from user input.

Also, this approach does not suffer from the ambiguity problems that container datatypes introduce. Consider a function for the equality of dynamic values, now using dynamic dictionary composition. We include duplicate contexts on purpose in the dynamic pattern matches:

```
dynEq :: Dynamic → Dynamic → Bool
dynEq (x :: a | Ord a) (y :: a | Ord a) = eq x y
dynEq ⊥ _ _ = ⊥
```

The crucial difference with container datatypes is that the contexts on *Ord* are part of the dynamic pattern matches, not of the dynamic values themselves. The same observation holds when the type of the dynamic value escapes to the context:

```
dynEq :: ∀ α . Dynamic → α → Bool | Ord α & TC α
dynEq (x :: a^ | Ord a^) y = eq x y
dynEq ⊥ _ = ⊥
```

We always refer to the same type class and the same instances. Therefore, no ambiguities can arise from duplicate contexts.

5. Ad-hoc polymorphism in dynamic typing

Now that we have seen how dynamic typing is included in the world of ad-hoc polymorphism, we continue by looking at the other way around: ad-hoc polymorphism in the world of dynamic typing. We identify two challenges and consider the sorting function from Section 2.1 as a running example in this section.

Naturally, the *sort* function is wrapped as follows, using the *wrap* function from Section 3.3:

```
wrappedSort :: Dynamic
wrappedSort = wrap sort
```

The first challenge is to come up with an appropriate type code, as dictated by the type of *wrap*.

One of the possibilities to unwrap values from a dynamic is by using the *unwrap* function from Section 3.3. For example, a function that unwraps and applies a value like *wrappedSort* is naively defined as follows:

```
dynAppOrd :: ∀ α . Dynamic → [α] → [α] | Ord α
dynAppOrd d x = unwrap d x
```

Unfortunately, this definition does not capture the intended behaviour. The type inferred for the result of unwrapping the value is too general; namely $\forall \alpha . [\alpha] \rightarrow [\alpha]$. Since we explicitly desire an ad-hoc polymorphic function, we have to provide an explicit type signature as well. In general, unwrapping ad-hoc polymorphic values always requires an explicit dynamic pattern match:

```

dynAppOrd :: ∀ α . Dynamic → [α] → [α] | Ord α
dynAppOrd (f :: ∀ α . [α] → [α] | Ord α) x = f x
dynAppOrd _ = ⊥

```

This function is ad-hoc polymorphic of its own; it has to propagate the *Ord* context to the unwrapped function. The first arm now includes an explicit dynamic pattern match that specifies an ad-hoc polymorphic type. The syntactic difference with the function *dynSort* from Section 4.2.1 is subtle: there we use a pattern variable while here we use a universally quantified type variable. The semantics of the former is opposite to the latter: there we have to produce an instance, while here we consume an instance. Here, the challenge is to extend existing semantics for unwrapping ad-hoc polymorphic values.

In this section we describe two different approaches to these challenges: dictionary-passing types (Section 5.1) and type code extension (Section 5.2).

5.1 Dictionary-passing types

The first approach makes clever use of the dictionary-passing style translation of type classes. This translation ‘removes’ ad-hoc polymorphism from a type, obtaining a parametric polymorphic type. Consider the dictionary-passing type of the function *sort*, as given before in Section 2.2:

```
sort :: ∀ α . DictOrd α → [α] → [α]
```

When wrapping such a value in a dynamic, a type code is required for its ad-hoc polymorphic type. However, we can make use of the existing type code mechanisms by requiring a type code for its dictionary-passing type instead.

Analogously, this approach translates dynamic pattern matches with an ad-hoc polymorphic type to a dictionary-passing type as well. For instance, the *dynAppOrd* function becomes:

```

dynAppOrd :: ∀ α . DictOrd α → Dynamic → [α] → [α]
dynAppOrd da (f :: ∀ α . DictOrd α → [α] → [α]) x =
  f da x
dynAppOrd da _ = ⊥

```

Since the type in the dynamic pattern match is no longer ad-hoc polymorphic, it only succeeds if it is provided a value that is exactly ad-hoc polymorphic in *Ord*: nothing more, nothing less.

5.1.1 Discussion

The advantage of the first approach is that it is lightweight: it is defined in terms of existing mechanisms. Consequently, the semantics of unwrapping ad-hoc polymorphic values does not take superclass relations into account. Also, this approach offers a backdoor. Since it translates dynamic pattern matches to include a dictionary-passing type, we can obtain the dictionary that is usually kept hidden from us. For example, when the expression **dynamic** ($\lambda da\ x \rightarrow x$) is presented as the first argument of *dynAppOrd*, the dynamic pattern match succeeds and the variable *da* is bound to the hidden internal dictionary. Although we cannot use the value since its dictionary type remains hidden, it is not very elegant.

5.2 Type code extension

The second approach relies less on existing mechanisms and extends these where necessary. As mentioned before, the use of *wrap* in the example function *wrappedSort* requires a type code for the ad-hoc polymorphic type of *sort*. The described type code definition in Section 3.4 extends naturally to include such types. Recall that we only consider type classes in the style of Haskell 98, where contexts in type signatures are of the form $C\ \alpha$ or $C\ (\alpha\ \tau_1 \dots \tau_n)$, where C is a type class, α a type variable, and τ_i is any type.

Then, we add a list of contexts to the *Scheme* alternative of the *TypeCode* type:

```

:: TypeCode = Scheme [String] TypeCode [Context]
              | ...
:: Context = Context ClassDef Parameter
:: Parameter = Parameter String [Type]
:: ClassDef
:: Type

```

Similar to type constructors, a context includes a class definition since name equivalence does not suffice. Also it contains a parameter, which is defined by a variable and a list of types. The list is empty for contexts of the form $C\ \alpha$. The definition of class definitions and types is left abstract. Then, the ad-hoc polymorphic type of sort $\forall \alpha . [\alpha] \rightarrow [\alpha] \mid Ord\ \alpha$ is represented as follows:

```

Scheme ["a"]
  (App (App (Con funDef)
            (App (Con listDef)
                  (Var "a"))))
      (App (Con listDef)
            (Var "a")))
[Context ordDef (Parameter "a" [])]

```

We leave the definition of the ordering type class, as well as the function and list type, abstract.

Since this approach extends the existing type code definitions, we have to extend the semantics for dynamic pattern matches involving such type codes as well. The proposed semantics as described before in Section 5.1 is unnecessarily restrictive. To illustrate this, we make a brief excursion to a similar phenomenon and consider rank-2 polymorphism (Odersky and Läufer, 1996; Peyton Jones et al., 2007). Suppose we define the rank-2 counterpart of *dynAppOrd* as follows:

```

rank2AppOrd ::
  ∀ α . (∀ α . [α] → [α] | Ord α) → [α] → [α] | Ord α
rank2AppOrd f x = f x

```

Here, we choose to lift the type from the first dynamic pattern match to the type of the function, replacing the occurrence of *Dynamic*. The first argument of the function is ad-hoc polymorphic and still expects a dictionary, which is provided by the context of *rank2AppOrd*. Evidently, we can omit the default case safely because the function no longer operates on dynamic values but on one specific lifted type. To gain insight in the desired semantics of dynamic pattern matches with ad-hoc polymorphic types, we look at possible arguments to *rank2AppOrd*. From less general to more general, we consider functions for sorting, removing duplicates from, and reversing lists:

```

sort    :: ∀ α . [α] → [α] | Ord α
nub     :: ∀ α . [α] → [α] | Eq α
reverse :: ∀ α . [α] → [α]

```

Clearly, providing the sorting function to *rank2AppOrd* is well typed; their types precisely match. Perhaps surprisingly, the duplicate removal function is suited as well; its type is more general than the sorting function since *Eq* is a superclass of *Ord*. Similarly, the reversing function poses no problem with the most general type of the three; it contains no contexts at all. Ideally, dynamic pattern matches with ad-hoc polymorphic types exhibit the same semantics. Then, the first arm of *dynAppOrd* must succeed for *sort*, *nub*, and *reverse* as well.

	Multi-parameter type classes	Flexible contexts	Flexible instances
Dynamic typing in ad-hoc polymorphism			
<i>Container datatypes</i>	●	○	○
<i>Dynamic dictionary composition</i>	●	●	●
Ad-hoc polymorphism in dynamic typing			
<i>Dictionary-passing types</i>	●	●	○
<i>Type code extension</i>	●	●	○

Figure 2. Overview of the interactions between approaches and type class extensions

5.2.1 Discussion

Opposite to the first approach, this approach is more heavyweight: it requires an extension of type codes and includes rank-2 polymorphism semantics in type unification. On the other hand, this approach does not provide any back doors and is therefore more elegant. Moreover, it is more flexible in dynamic pattern matches. However, it can be desirable to precisely pattern match on contexts, not taking superclass relations into account. Luckily, such behaviour is achieved by enumerating the cases from more general to less general. For example, we distinguish the three function types of *sort*, *nub*, and *reverse* by the following ordering of dynamic pattern matches:

```

distinguish :: Dynamic → ...
distinguish (f :: ∀ α . [α] → [α]) = ...
distinguish (f :: ∀ α . [α] → [α] | Eq α) = ...
distinguish (f :: ∀ α . [α] → [α] | Ord α) = ...
distinguish _ = ⊥

```

The first arm matches only *reverse*, the second arm matches *nub* as well, and the final arm matches all three functions. Consequently, if we wish to distinguish all n superclasses of a type class, this results in $n + 1$ arms.

6. Type class extensions

Until now, we have only considered the realisation of ad-hoc polymorphism through type classes in the style of Haskell 98. In this section we discuss some of the more popular type class extensions (Peyton Jones et al., 1997) in Clean and Haskell: multi-parameter type classes (Section 6.1), flexible contexts (Section 6.2), and flexible instances (Section 6.3). We give a brief introduction to each extension and discuss if and how it affects dynamic typing in ad-hoc polymorphism, as in Section 4, and ad-hoc polymorphism in dynamic typing, as in Section 5. Not all of the described approaches are affected by all extensions though. Figure 2 provides additional guidance for this section by summarising the interactions that require discussion.

6.1 Multi-parameter type classes

We assumed in earlier sections that the number of type class parameters is restricted to one. The multi-parameter type class extension lifts the restriction so that a type class can have any number of parameters, which do not necessarily need to be distinct variables. Consider the following multi-parameter type class, one that models an array of type α with elements of type ϵ , with a single member function that returns the value at the indicated position:

```

class Array α ε where
  select :: Int → α ε → ε

```

Then, an instance for lists of integers is defined as follows:

```

instance Array [] Int where
  select i x = ...

```

The corresponding dictionary type now takes two parameters:

```

:: DictArray α ε = { select :: Int → α ε → ε }
dictArrayListInt :: DictArray [] Int
dictArrayListInt = { select = λi x → ... }

```

Evidently, also the form of contexts occurring in type signatures change accordingly. For example in the function that selects the first element of an array:

```

selectFirst :: ∀ α ε . α ε → ε | Array α ε
selectFirst = select 0

```

Since the complete form of type classes is affected by this extension, dynamic typing in ad-hoc polymorphism as well as ad-hoc polymorphism in dynamic typing is affected.

6.1.1 Dynamic typing in ad-hoc polymorphism

Similar to *dynSort* as defined in Section 4, we naively would define a dynamic function that selects the first element of an array as follows:

```

dynSelectFirst :: Dynamic → Dynamic
dynSelectFirst (x :: a e) = dynamic (selectFirst x)
dynSelectFirst _ = ⊥

```

Again, type information that is required to resolve ad-hoc polymorphism only becomes available at run time. We describe how both container datatypes and dynamic dictionary composition are extended to support resolving of multi-parameter type classes.

Container datatypes The extension is incorporated naturally in container datatypes. For instance, when we define a container that captures values together with their *Array* instance:

```

:: ContArray α ε = ContArray ((α ε) | Array α ε)

```

The container datatype now takes two parameters, in contrast to the similar definition of *ContOrd* from Section 4.1. Then, we adapt the function to include the constructor of the container datatype:

```

dynSelectFirst :: Dynamic → Dynamic
dynSelectFirst (ContArray x da :: ContArray a e) =
  dynamic (selectFirst x)
dynSelectFirst _ = ⊥

```

As before, the container datatype carries an additional field in dictionary-passing style:

```

:: ContArray α ε = ContArray (α ε) (DictArray α ε)

```

Then, the *dynSelectFirst* function makes the dictionary available in the dynamic pattern match:

```

dynSelectFirst :: Dynamic → Dynamic
dynSelectFirst (ContArray x da :: ContArray a e) =
  dynamic (selectFirst da x)
dynSelectFirst _ = ⊥

```

The obtained dictionary is passed on to the *selectFirst* function.

We leave its definition abstract since this would require a more elaborate *Array* type class with more member functions.

Both dynamic typing in ad-hoc polymorphism as well as ad-hoc polymorphism in dynamic typing are affected by flexible contexts. The former since the form of contexts in instances is changed and the latter since the form of contexts in type signatures is changed.

6.2.1 Dynamic typing in ad-hoc polymorphism

We only have to consider dynamic dictionary composition since container datatypes are not concerned with more flexible contexts in class definitions, instance definitions, or type signatures.

Dynamic dictionary composition Consider the earlier introduced generator function for the *Array* type class from Section 6.1.1. The instance for lists and lists of values, defined using the flexible contexts extension, adds the following arm:

$$\begin{aligned} \text{genDictArray } (_ :: []) (_ :: [e]) = \\ \text{do } da \leftarrow \text{genDictArray } (\text{dynamic } \perp :: []) \\ (\text{dynamic } \perp :: e) \\ \text{Just } (\text{dynamic } (\text{dictArrayListList } (\text{unwrap } da))) \end{aligned}$$

Here, the recursive call to the generator function no longer just takes pattern variables obtained from the dynamic pattern match, but ordinary types as well, as visible in its first argument. The expressive dynamic typing system allows us to construct a dynamic value of any type, therefore, this extension is easily taken care of. Note that if we lifted the coverage conditions as mentioned earlier, termination of the generator function is not guaranteed.

6.2.2 Ad-hoc polymorphism in dynamic typing

A value like *concatArray* is wrapped as usual:

$$\begin{aligned} \text{wrappedConcatArray} :: \text{Dynamic} \\ \text{wrappedConcatArray} = \text{wrap } \text{concatArray} \end{aligned}$$

Again, such a value is unwrapped by explicating its type in a dynamic pattern match:

$$\begin{aligned} \text{dynAppArray} :: \forall \epsilon . \text{Dynamic} \rightarrow [[\epsilon]] \rightarrow [\epsilon] \mid \text{Array } [] \epsilon \\ \text{dynAppArray } (f :: \forall \epsilon . [[\epsilon]] \rightarrow [\epsilon] \mid \text{Array } [] \epsilon) x = f x \\ \text{dynAppArray } _ \quad \quad \quad _ = \perp \end{aligned}$$

Since the form of contexts in type signatures is changed by the extension, both dictionary-passing types and type code extension are affected.

Dictionary-passing types Flexible contexts are straightforwardly included in the dictionary-passing types approach. For instance, when we wrap the *concatArray* function, a type code is included for the following type:

$$\text{concatArray} :: \forall \epsilon . \text{DictArray } [] \epsilon \rightarrow [[\epsilon]] \rightarrow [\epsilon]$$

As before, the dictionary-passing type is used in the dynamic pattern match as well:

$$\begin{aligned} \text{dynAppArray} :: \\ \forall \epsilon . \text{DictArray } [] \epsilon \rightarrow \text{Dynamic} \rightarrow [[\epsilon]] \rightarrow [\epsilon] \\ \text{dynAppArray } da (f :: \forall \epsilon . \text{DictArray } [] \epsilon \rightarrow [[\epsilon]] \rightarrow [\epsilon]) \\ \quad \quad \quad x = f da x \\ \text{dynAppArray } da \quad \quad \quad _ = \perp \end{aligned}$$

Note that the dynamic pattern match not only succeeds any more for values that are exactly ad-hoc polymorphic in *Array*. Since the first parameter of the dictionary type is restricted to the list type, any dictionary that is less restrictive will also do.

Type code extension We adapt the *Parameter* type, that models type class parameters, from Section 5.2 to include flexible contexts:

$$:: \text{Parameter} = \text{Parameter } [Type]$$

It now includes a list of types, instead of always requiring a type variable in prefix position. Again, the *ClassDef* type has to be modified as well to include the new form of contexts in class definitions. Unwrapping values including flexible contexts follow rank-2 polymorphism semantics.

6.3 Flexible instances

Besides a restricted context, we also restricted the instances to be of the form $C (T \alpha_1 \dots \alpha_n)$ where C is a type class, T a type constructor, and α_i a distinct type variable. The flexible instances extension lifts this restriction, including multi-parameter type classes, to the form $C \tau_1 \dots \tau_n$, where C is a type class and τ_i is any type. For example, we define an instance of *Array* for lists and pairs with integers:

$$\begin{aligned} \text{instance } \text{Array } [] (Int, \epsilon) \text{ where} \\ \text{select } i x = \dots \end{aligned}$$

The corresponding dictionary definition reflects the flexible instance in its type:

$$\begin{aligned} \text{dictArrayListPairInt} :: \forall \epsilon . \text{DictArray } [] (Int, \epsilon) \\ \text{dictArrayListPairInt} = \{ \text{select} = \lambda i x \rightarrow \dots \} \end{aligned}$$

Due to the extension, overlap between instances can occur. For example, consider the following additional instance:

$$\begin{aligned} \text{instance } \text{Array } \alpha (Int, \epsilon) \mid \text{Array } \alpha \epsilon \text{ where} \\ \text{select } i x = \dots \end{aligned}$$

Here, the instances overlap in the first parameter of *Array*. Consequently, it is not clear which instance to choose for an array whose elements are of type $(Int, Bool)$. Normally, such ambiguities are rejected at compile time. Another extension called overlapping instances lifts this restriction and chooses the most specific one. In this example, the first one is most specific. Evidently, there is not always such an instance, consider for example the following:

$$\begin{aligned} \text{instance } \text{Array } [] (\epsilon, Bool) \text{ where} \\ \text{select } i x = \dots \end{aligned}$$

Then, the overlapping instances are rejected at compile time.

Since the extension only affects the form of instances, only dynamic typing in ad-hoc polymorphism is affected.

6.3.1 Dynamic typing in ad-hoc polymorphism

Only dynamic dictionary composition is concerned with the form of instances. Therefore, we do not consider container datatypes.

Dynamic dictionary composition As before, each instance of a type class results in an arm of the corresponding generator function. The instance of *Array* for lists and pairs with integers gives the following arm:

$$\begin{aligned} \text{genDictArray } (_ :: []) (_ :: (Int, e)) = \\ \text{Just } (\text{dynamic } \text{dictArrayListPairInt}) \end{aligned}$$

The more flexible form of the second parameter results in a more elaborate dynamic pattern match. Again, the dynamic typing system is expressive enough to cope with such types in a dynamic pattern match.

Overlapping instances are resolved at compile time. If there is not a single most specific instance to choose, a compile-time error occurs. In our approach we use a generator function to compose dictionaries. Therefore, it is only until run time that we are able to verify this condition. Unfortunately, the generator function is mechanically deduced at compile time, choosing an explicit ordering of the arms. Consequently, overlapping instances are not supported in this approach.

7. Related work

An extensive overview of the interaction between ad-hoc polymorphism and dynamic typing in a statically typed functional language has not been described earlier. However, bringing the worlds of ad-hoc polymorphism and dynamic typing has been recognised before by Plasmeijer and van Weelden (2005). An interactive shell is described to interpret user-provided values using the dynamic typing system. To facilitate ad-hoc polymorphism, the dictionary-passing style is made explicit when translating the value provided by the user to an internal structure. Unfortunately, this is restricted to pre-defined type classes; additional instances cannot be provided by the user. Evidently, the approaches described in this paper are not restricted in that sense. Furthermore, our approach is more flexible since we are not confined to the world of dynamic typing.

Before type classes, Kaes (1988) already described an approach towards ad-hoc polymorphism named parametric overloading where functions are parameterised with additional arguments that represent the abstracted behaviour. In that sense, these functions are not ad-hoc polymorphic but parametric polymorphic. Wadler and Blott (1989) improved on this technique by allowing the additional parameters to be grouped in type classes, and described the translation to dictionary-passing style which we used extensively in this paper. However, the approach of parametric overloading did include a mechanism that resolves ad-hoc polymorphism dynamically. Unfortunately, this requires the additional parameters (i.e., the dictionary) to be strict, and possibly resulted in nontermination. Our approaches of container datatypes and dynamic dictionary composition do not use parametric overloading but include the full power of type classes without compromising laziness nor termination.

Leroy and Mauny (1993) describe dynamic typing with parametric polymorphism and show how this is used to model ad-hoc polymorphism. Functions enumerate all possible expected types using dynamic values, which is named structural ad-hoc polymorphism. Its opposite is embodied by type classes and is called nominal ad-hoc polymorphism. The main difference is that the nominal variant is ‘open’ (i.e., instances can be given anywhere), whereas the structural variant is ‘closed’ (i.e., every ad-hoc polymorphic function enumerates the possible cases). The former is orthogonal to the latter: instances do not require exhaustive enumerations, though their definitions are dispersed. A unification of both variants in a single functional language is described elsewhere (Vytiniotis et al., 2004). Our dynamic dictionary composition approach uses both: dispersed instances are mechanically grouped at compile time in a single generator function to capture all available instances.

While we considered statically typed functional languages like Clean, Haskell, and ML, other functional languages that are dynamically typed also support ad-hoc polymorphism. For instance, languages like Lisp and Scheme resolve ad-hoc polymorphism at run time since only then type information becomes available. While these languages use a similar dispatching mechanism like the generator function, there is no support for an expressive static system like type classes.

We describe run-time resolving of ad-hoc polymorphism that can fail, consistent with the original semantics of dynamic pattern matches. Rouaix (1990) describes an approach, inspired by object-oriented languages, where a restricted form of ad-hoc polymorphism is resolved at run time without any possibility of run-time failure. However, this is described in a statically typed language, while our approaches especially consider such languages that support dynamic typing as well.

Object-oriented languages, being statically typed like Java and Scala or dynamically typed like Smalltalk, resort to run-time resolving of ad-hoc polymorphism due to their late binding. Only at run time it can be determined which method is used.

8. Conclusion

We have given an elaborate overview of the interaction between ad-hoc polymorphism and dynamic typing in a statically typed functional language. We identified two sides to their interaction: dynamic typing in ad-hoc polymorphism and ad-hoc polymorphism in dynamic typing, introducing one world into the other. Regarding the former interaction, we showed two complementary approaches, namely container datatypes and dynamic dictionary composition, that provide mechanisms to resolve ad-hoc polymorphism depending on dynamic type information. Both approaches are best suited in different applications, either requiring rigidity or flexibility. Also, both approaches can happily coexist. With respect to the latter interaction, we showed two different approaches, namely dictionary-passing types and type code extension, to wrap and unwrap ad-hoc polymorphic values in a dynamic. These approaches differ in implementation effort and flexibility of pattern-matching semantics. Finally, we discussed several type class extensions and argued that most of these fit naturally in the described mechanisms. Only lifting the restrictions of flexible contexts using undecidable instances and flexible instances using overlapping instances are not supported by dynamic dictionary composition.

Some of the work described in this paper has been experimentally included in Clean: container datatypes that expose the type of their content, as well as the possibility to (un)wrap ad-hoc polymorphic values via dictionary-passing types. We plan to experiment with the other approaches in Clean as well. Also, we aim to further investigate the relation between ad-hoc polymorphism and dynamic typing via a more formal approach to their interactions.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. The authors are indebted to John van Groningen for the original idea of dictionary-passing types and the helpful discussions on the related subjects. This work has been funded by the Technology Foundation STW through its project on “Demand Driven Workflow Systems” (07729).

References

- Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.
- Martín Abadi, Luca Cardelli, Benjamin Pierce, Didier Rémy, and Robert Taylor. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):81–110, 1994.
- Peter Achten. Clean for Haskell 98 programmers - A quick reference guide. <http://www.st.cs.ru.nl/papers/2007/achp2007-CleanHaskellQuickGuide.pdf>, 2007.
- Arthur Baars and Doaitse Swierstra. Typing dynamic typing. In Simon Peyton Jones, editor, *Proceedings of the 7th International Conference on Functional Programming, ICFP '02, Pittsburgh, PA, USA*, pages 157–166. ACM Press, 2002.
- James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel Chakravarty, editor, *Proceedings of the 6th Haskell Workshop, Haskell '02, Pittsburgh, PA, USA*, pages 90–104. ACM Press, 2002.
- John van Groningen, Thomas van Noort, Peter Achten, Pieter Koopman, and Rinus Plasmeijer. Exchanging sources between Clean and Haskell - A double-edged front end for the Clean compiler. In Jeremy Gibbons, editor, *Proceedings of the 3rd Haskell Symposium, Haskell '10, Baltimore, MD, US*. ACM Press, 2010. To appear.
- Stefan Kaes. Parametric overloading in polymorphic programming languages. In Harald Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming, ESOP '88, Nancy, France*, pages 131–144. Springer-Verlag, 1988.

- Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.
- Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.
- Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: the end of the cold war between programming languages. In Roel Wuyts, editor, *Proceedings of the 1st International Workshop on Revival of Dynamic Languages, OOPSLA '04, Vancouver, BC, Canada*. ACM Press, 2004.
- Martin Odersky and Konstantin Läufer. Putting type annotations to work. In Hans Boehm, editor, *Proceedings of the 23rd Symposium on Principles of Programming Languages, POPL '96, St. Petersburg Beach, FL, USA*, pages 54–67. ACM Press, 1996.
- John Peterson and Mark Jones. Implementing type classes. In Robert Cartwright, editor, *Proceedings of the 6th Conference on Programming Language Design and Implementation, PLDI '93, Albuquerque, NM, USA*, pages 227–236. ACM Press, 1993.
- Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.
- Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In John Launchbury, editor, *Proceedings of the 2nd Haskell Workshop, Haskell '97, Amsterdam, The Netherlands*. ACM Press, 1997.
- Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In Julia Lawall, editor, *Proceedings of the 11th International Conference on Functional Programming, ICFP '06, Portland, OR, USA*, pages 50–61. ACM Press, 2006.
- Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.
- Simon Peyton Jones, Dimitrios Vytiniotis, Tom Schrijvers, and Martin Suzmann. Modular type inference with local assumptions. *Journal of Functional Programming*, 2010. Under consideration for publication.
- Marco Pil. First class file I/O. In Chris Clack, Kevin Hammond, and Antony Davie, editors, *Selected Papers of the 8th International Workshop on Implementation of Functional Languages, IFL '97, St. Andrews, UK*, volume 1467 of *Lecture Notes in Computer Science*, pages 233–246. Springer-Verlag, 1997.
- Marco Pil. Dynamic types and type dependent functions. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Selected Papers of the 10th International Workshop on the Implementation of Functional Languages, IFL '98, London, UK*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer-Verlag, 1999.
- Rinus Plasmeijer and Arjen van Weelden. A functional shell that operates on typed and compiled applications. In Varmo Vene and Tarmo Uustalu, editors, *Proceedings of the 5th International Summer School on Advanced Functional Programming, AFP '04, Tartu, Estonia*, volume 3622 of *Lecture Notes in Computer Science*, pages 245–272. Springer-Verlag, 2005.
- Francois Rouaix. Safe run-time overloading. In Frances Allen, editor, *Proceedings of the 17th Symposium on Principles of Programming Languages, POPL '90, San Francisco, CA, US*, pages 355–366. ACM Press, 1990.
- Martin Sulzmann, Gregory Duck, Simon Peyton Jones, and Peter Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17(1):1–82, 2007.
- Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In Greg Morrisett and Manuel Fähndrich, editors, *Proceedings of the 5th Workshop on Types in Language Design and Implementation, TLDI '05, Long Beach, CA, USA*, pages 13–24. ACM Press, 2004.
- Phil Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th Symposium on Principles of Programming Languages, POPL '89, Austin, TX, US*, pages 60–76. ACM Press, 1989.
- Stefan Wehr and Manuel Chakravarty. ML modules and Haskell type classes: a constructive comparison. In Ganesan Ramalingam, editor, *Proceedings of the 6th Asian Symposium on Programming Languages and Systems, APLAS '08, Bangalore, India*, volume 5356 of *Lecture Notes in Computer Science*, pages 188–204. Springer-Verlag, 2008.
- Ronny Wichers Schreur and Rinus Plasmeijer. Dynamic construction of generic functions. In Clemens Grelck, Frank Huch, Greg Michaelson, and Phil Trinder, editors, *Revised Selected Papers of the 16th International Workshop on the Implementation and Application of Functional Languages, IFL '04, Lübeck, Germany*, volume 3474 of *Lecture Notes in Computer Science*, pages 160–176. Springer-Verlag, 2005.