

Typing Dynamic Typing

Arthur I. Baars
arthurb@cs.uu.nl

S. Doaitse Swierstra
doaitse@cs.uu.nl

Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80.089
3508 TB Utrecht, The Netherlands

Abstract

Even when programming in a statically typed language we every now and then encounter statically untypable values; such values result from interpreting values or from communicating with the outside world. To cope with this problem most languages include some form of *dynamic* types. It may be that the core language has been explicitly extended with such a type, or that one is allowed to live dangerously by using functions like *unsafeCoerce*. We show how, by a careful use of existentially and universally quantified types, one may achieve the same effect, without extending the language with new or unsafe features. The techniques explained are universally applicable, provided the core language is expressive enough; this is the case for the common implementations of Haskell. The techniques are used in the description of a type checking compiler that, starting from an expression term, constructs a typed function representing the semantics of that expression. In this function the overhead associated with the type checking is only once being paid for; in this sense we have thus achieved static type checking.

Categories and Subject Descriptors

D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.3 [Programming Languages]: Language Constructs and Features—*abstract data types, polymorphism, control structures*; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*control primitives, functional constructs, type structure*

General Terms

Languages, Theory

Keywords

Dynamic typing, static typing, type equality, coercions, quantified types, Leibnitz' rule, Haskell, typed interpreters.

1 Introduction

For a statically typed programming language the typing rules are checked at compile-time. Type violations are reported before program execution, and efficient object code can be generated since no type consistency checks have to be performed at run-time. Even when using statically typed languages however, the need arises to deal with values with types that cannot be determined at compile-time. This situation occurs for example in a program that interprets a (meta)language term, resulting in an object language value of a type that depends on the specific term at hand. An example of such an interpretation function is *eval* which takes a string, parses it into an expression and returns the value of that expression. Other examples where type information is not available until run-time are distributed programs, that exchange data between different processes, and programs that store a value of an arbitrary type in, and retrieve it from stable storage.

There exist several proposals [1, 2, 8] for dealing with such dynamically typed values in a statically typed language. These are all based on a similar idea: *extend the language* with a universal type *dynamic*, and embed dynamic values in that type. This new type actually is a pair consisting of the value itself, together with a representation of the type of that value. Before using a dynamic value its type component is inspected using a *typecase* or similar construct.

The proposals differ in their capability to deal with polymorphic dynamic values. In [1] dynamic polymorphism is forbidden, for [8] a restricted form of dynamic polymorphism is allowed and the system described in [2] allows polymorphic values with little restrictions, and as to be expected is also the most complex one. The latest release of Clean [12] implements support for dynamic typing [11] by providing a *typecase* construct and allowing polymorphic dynamic values. A drawback noted by Shields *et al.* [13] is that types live in two different worlds, with explicit conversions between these two worlds; he views dynamic typing as staged type inference. Some program expressions thus have their type inference deferred until enough information is available at run-time.

Despite ongoing research on dynamic typing, many statically typed languages – such as Haskell [9] – do not have built-in support for dynamic typing. A programmer thus must encode the dynamic type tagging and type checks explicitly. If the set of types is finite and known at compile-time, one may decide to embed values in a user-defined data type, and subsequently inspect the types of a values using case-analysis. With this approach, however, programs are difficult to maintain and become verbose and messy because type-checking code is intertwined with normal code.

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.
ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.
Copyright 2002 ACM 1-58113-487-8/02/0010 ...\$5.00

To alleviate this problem the distributions of both Hugs and GHC provide a dynamic typing library. This library provides a universal representation *Dynamic* for dynamically typed values. Furthermore it provides a function *toDyn* that injects a value of arbitrary type into a dynamically typed value and a function *fromDyn* that converts a dynamic value into a concrete (monomorphic) type. Although this library cannot deal with polymorphic dynamic values, it has proven to be quite useful in practice. It provides a cheap way to program with dynamically typed values in Haskell. The implementation of this library is fairly simple and is based on the function *unsafeCoerce* which has type $(\forall ab. a \rightarrow b)$. As the name suggests this function is not type-safe, a direct consequence of the fact that safe functions of type $(\forall ab. a \rightarrow b)$ cannot exist at all.

In this paper we develop a statically typable library that provides a form of dynamic typing, that is as powerful as Haskell’s Dynamic library, but does not make use of unsafe functions or compiler extensions. The attractive aspect of this approach is that we do not extend the language itself, nor its implementation. Thus the technique can be universally applied, provided the type system of the host language is powerful enough to type our library, as is the case for Haskell with its common extensions for existentially and universally quantified types and constructors.

This paper is organized as follows: Section 2 introduces the concepts of dynamic typing. In Section 3 a data type is developed that serves as a witness of a proof that two types are equal. Section 4 defines a set of proof combinators to easily construct inhabitants of this data type. A key ingredient for dynamic typing are run-time type tags, and thus in Section 5 a class of type representations is introduced and a typical instance of this class is given. In Section 6 an interpreter for expression terms is described as an application of the techniques developed in this paper. Section 7 concludes.

2 Dynamic Typing

Similar to Haskell’s dynamic library and other approaches to dynamic typing we take the data type *Dynamic* to be a pair of a value together with the representation of its type. Suppose we have a type *typeRep* that is used to represent types; then the question arises how we can convince the type checker of our core language (in our case Haskell), that a value of type *typeRep* corresponds to a specific Haskell type; i.e. how can we label such a type representation with the type it represents. The answer lies in using a type constructor instead of a simple type, i.e. by passing the label type as an argument to the representation of the type.

A dynamic value can be viewed as a box containing a value of some type and the representation of that type. Since we do not yet want to fix the type we use for representing types, we postpone this decision by making this *typeRep* a parameter of the data type *Dynamic*. The actual type of the value injected in a *Dynamic* is hidden by existentially quantifying¹ over its type. Note that $::$ is an infix constructor function with two arguments, namely an *a* and a *typeRep a*.

```
data Dynamic typeRep =  $\exists a. a :: typeRep a$ 
```

Using the $::$ constructor a value can be packed as a dynamic, hiding the actual type of the value.

¹The implementors of the Haskell extensions have chosen to denote this existential quantifier with a **forall** keyword. In this presentation we have taken the liberty to use a somewhat more intuitive notation using the \exists symbol

```
toDynamic      :: a  $\rightarrow$  tpr a  $\rightarrow$  Dynamic tpr
toDynamic a tpr_a = a :: tpr_a
```

A value with an existential type can be unpacked using a case-construct.

```
case dynval of
  a :: tpr_a  $\rightarrow$  ...
```

A skolem constant is introduced to serve as a placeholder for the type of *a* and the label of the type of the type representation *tpr*. Unfortunately, the value *a* cannot simply be returned, because then the skolem constant representing the type of *a* would escape, i.e. appear in a type outside the scope of the case-expression.

We thus need a partial function *fromDyn* that converts a dynamic value into a value with a concrete type:

```
fromDyn :: typeRep a  $\rightarrow$  Dynamic typeRep  $\rightarrow$  Maybe a
```

The function *fromDyn* takes two arguments, the representation of the expected type and a dynamic value. Dynamic type checking now boils down to comparing the type representation of the expected type, say *t1* of type *tp a* and the representation of the dynamic value’s type, say *t2* of type *tp b*, and to make sure that the following equivalence holds: $(t1 \sim t2) \leftrightarrow (a \equiv b)$, i.e. structural equivalence of two type representations implies type equivalence of their labels and vice versa. Since the right to left direction is not used we will focus on the left to right direction of this implication, i.e. on $(t1 \sim t2) \rightarrow (a \equiv b)$. The function *fromDyn* must compare the representations of the expected type and the dynamic’s type tag. If they are equal then the value stored in the dynamic is returned as a value of the expected type. Unfortunately simply returning the value stored in the dynamic will not work: the type of this value is unknown since it is still shielded by the existential quantification. The question thus arises how to convince Haskell’s type-checker of the fact that a successful comparison implies that the hidden type equals the expected type, and hence is not unknown.

3 Type Equality

In this section we develop a data type that serves as evidence for the type-checker that two types are equal.

In a first approach we consider two types equivalent if the first can be converted into the second and vice versa, as expressed in by the type *Equal*:

```
type Equal a b = (a  $\rightarrow$  b, b  $\rightarrow$  a)
```

This encoding of type equivalence is used among others by Yang[17] and Weirich[16]. Interestingly, this type is the encoding, according the Curry-Howard isomorphism of the following definition of equivalence on propositions. Note that function arrows map to implications and pairs to conjunctions.

```
a  $\equiv$  b = a  $\rightarrow$  b  $\wedge$  b  $\rightarrow$  a
```

The Curry-Howard isomorphism concerns the correspondence between logical formulas and types, in which a logical proof corresponds to a computational term. The existence of a (non-diverging) term of a certain type implies the existence of a proof of the corresponding logical formula, and vice versa. Hence the existence of an embedding-projection pair of type *Equal a b* can be considered a proof that *a* and *b* are equivalent.

Unfortunately the above solution fails to enforce that the existence

of a value of type *Equal a b* implies that *a* and *b* are truly equal, as can be seen in the following example:

```
eqIntBool      :: Equal Int Bool
eqIntBool      = (even, λx → if x then 1 else 0)
```

The problem is that arbitrary functions can be chosen as conversion function. So we want to restrict the type *Equal* in such a way that it is ensured that it can only contain conversion functions that leave the dynamic value unchanged. This can be achieved by “shielding” the types *a* and *b* using a universally² quantified type constructor *f*:

```
newtype Equal a b = Equal (∀ f. f a → f b)
```

This encoding of type equality, in a more implicit way, also serves as a basis for the implementation of a type-safe cast in Weirich[16].

Interestingly, the data type *Equal* is actually the encoding of Leibnitz’ law which states that if *a* and *b* are identical then they must have identical properties. Leibnitz’ original definition reads as follows

$$a \equiv b \quad = \quad \forall f. f a \Leftrightarrow f b$$

and can be proven to be equivalent to:

$$a \equiv b \quad = \quad \forall f. f a \rightarrow f b$$

The *Equal* data type encodes true type equality, since the identity function is the only non-diverging conversion function that can be used as argument of the *Equal* constructor. As the conversion function has to work for any *f*, it cannot make assumptions about the structure of *f*, making it impossible to construct a value of type *f a* or to access values of type *a* that may be stored inside a value of type *f a*. Hence it is impossible for a conversion function to alter the value it takes as argument. Not taking into account the failing functions \perp and $\lambda x \rightarrow \perp$, the identity function is the only function that can be used to construct a value of type *Equal*. The existence of a value of type *Equal a b* now implies that $a \equiv b$, since the conversion function, that converts an *a* into a *b*, must be the identity function.

In the definition of *Equal* the kinds of the type variables *a* and *b* can not be determined. Instead of defaulting these kinds to *, as is done in Haskell, we assume our language supports polymorphic kinds and assign the following kind to the type constructor *Equal*:

$$Equal : \forall k. k \rightarrow k \rightarrow *$$

Kind polymorphism would make a language such as Haskell more powerful and flexible. Consider, for example, the following type constructors *Int*, *[]* and \rightarrow which have kinds *, $* \rightarrow *$, and $* \rightarrow * \rightarrow *$ respectively. Because of the polymorphic kind the types *Equal Int Int*, *Equal [] []* and *Equal (\rightarrow) (\rightarrow)* are all valid, whereas Haskell with its defaulting rule only accepts the first.

4 Building Equalities

Values of type *Equal a b* can be viewed as a proof that the types *a* and *b* are equal. This section introduces a set of proof combinators that can be used to easily construct equality proofs, and thus to create inhabitants of the *Equal* type.

We continue by showing that the type *Equal* implements an equiv-

²Note that we have chosen to write the Haskell keyword **forall** as a \forall .

alence relation on types. The properties of an equivalence relation, namely reflexivity, transitivity and symmetry, can be encoded as:

Reflexivity ($a \equiv a$):

```
reflex          :: Equal a a
reflex          = Equal id
```

Transitivity ($a \equiv b \wedge b \equiv c \Rightarrow a \equiv c$):

```
trans          :: Equal a b → Equal b c → Equal a c
trans ab bc    = case (ab, bc) of
                 (Equal f, Equal g) → Equal (g.f)
```

The implementation of *symmetry* ($a \equiv b \Rightarrow b \equiv a$) is based on the following idea. If *a* and *b* are equal, we can freely substitute *a*’s in a term by *b*’s. Assuming $a \equiv b$ we can derive that $b \equiv a$ by starting with $a \equiv a$ (reflexivity) and applying the substitution $a \mapsto b$ on the first *a*.

The question that arises is how to implement such a substitution. Recall that $a \equiv b$ is encoded by a transform function of type $\forall f. f a \rightarrow f b$. It substitutes the argument *a* of a type constructor by *b*. In order to do a substitution on the *a*’s in a type *t*, we first introduce a type constructor *c*, that is the abstraction of *t* over the *a*’s we wish to substitute. The type *t* is equivalent to *c a*, applying the transformation function yields a result of type *c b*, which is the type *t* with the *a*’s substituted by *b*’s. For example we wish to apply the substitution $a \mapsto b$ on the type (a, a) . This type is equivalent to the application $(\lambda x. (x, x))a$. Applying the transform function yields a result of type $(\lambda x. (x, x))b$, which can then be reduced to (b, b)

In Haskell lambda abstraction on type level can be mimicked using data types. The data type corresponding to the tuple example above is:

```
newtype Pair x = Pair { unPair :: (x, x) }
```

A value of type (a, a) can be tagged with the *Pair* constructor to view its type as the application *Pair a*. After applying the transformation function, the reduction of the type *Pair b* to (b, b) is done by untagging the value. The function *substPair* is the implementation of the substitution $a \mapsto b$ on the type (a, a) . It uses the function *subst*, that takes care of the tagging and untagging and performs the actual substitution by applying the transformation function *ab*.

```
subst          :: (ta → ca)
               → (cb → tb)
               → Equal a b
               → ta
               → tb
subst from to (Equal ab) = to.ab.from

substPair      :: Equal a b → (a, a) → (b, b)
substPair      = subst Pair unPair
```

We will use the term type combinator for type constructors such as *Pair*. A type combinator constructs a type (in this case (a, a)) from its components (in this case *a*). The last argument of a type combinator is the target of the substitution. The others (if any) are unaffected by the substitution. Consider for example the type combinator *Middle*, which combines three component types into a triple:

```
newtype Middle x y z = Middle { unMiddle :: (x, z, y) }
substMiddle          :: Equal a b → (x, a, y) → (x, b, y)
substMiddle          = subst Middle unMiddle
```

As the middle component of the $\text{triple}(z)$ is the last argument of the type combinator it is the target of the substitution, whereas the first and the third component are unaffected. Any substitution can be expressed using the function *subst* and the appropriate type combinator.

Finally we have all the ingredients to implement the symmetry property for the *Equal* type. The idea is that *Equal b a* can be derived from *Equal a b* and reflexivity (*Equal a a*) by applying the substitution $a \mapsto b$ to the first *a* in *Equal a a*. Firstly we define a type combinator *FlipEqual* to allow a substitution on the first argument of the *Equal* type constructor:

```
newtype FlipEqual y x = Flip{unFlip :: Equal x y}
```

The type variable *x* represents the target of the substitution since it is the last argument of *FlipEqual*. The symmetry property can now straightforwardly expressed:

Symmetry ($a \equiv b \Rightarrow b \equiv a$):

```
symm          :: Equal a b → Equal b a
symm ab       = (subst Flip unFlip ab) reflex
```

The functions *reflex*, *trans*, and *symm* are proof combinators, they implement proof rules, and construct proofs out of other proofs. In the remainder of this section we extend our library of proof combinators with other rules. Substitutions play a key role in the implementation of these rules.

We proceed by introducing the following rule:

$$\text{ARG} : \frac{a \equiv b}{f a \equiv f b}$$

This rule is implemented by the function *arg*, which has the following type:

```
arg          :: Equal a b → Equal (f a) (f b)
```

Notice that *Equal (f a) (f b)* is encoded as a function with the following type ($\forall g.g (f a) \rightarrow g (f b)$). Thus the function *arg* can be implemented as a substitution on the *a* in the type *g (f a)*. The type combinator, that is required for this substitution, is the following:

```
newtype Comp g f x = Comp{unComp :: g (f x)}
```

Using this type combinator, the function *arg* is now defined as:

```
arg ab       = Equal (subst Comp unComp ab)
```

A very useful proof combinator derived from *arg* is *rewrite*. It can be used to apply a substitution to the argument of a type constructor. For example if we have two proofs $x :: \text{Equal } a \text{ Int}$ and $list :: \text{Equal } b [a]$ then we can construct a proof of type *Equal b [Int]* as follows: *rewrite x list*.

```
rewrite      :: Equal a b
             → Equal c (f a)
             → Equal c (f b)
rewrite a_b c_fa = let fa_fb = arg a_b
                  in trans c_fa fa_fb
```

The function *rewrite* can only apply a substitution to the last argument of a type constructor, such as the argument of the list type constructor or the result part of the function arrow. We now formu-

late the corresponding rule for the function part:

$$\text{FUNC} : \frac{f \equiv g}{f a \equiv g a}$$

The type of the function implementing this rule is:

```
func          :: Equal f g → Equal (f a) (g a)
```

Note that a value of type *Equal (f a) (g a)* is represented as a function with the following type: $\forall h.h (f a) \rightarrow h (g a)$. Hence a substitution must be applied on the *f* in the type *h (f a)*. This substitution is again implemented using the function *subst* and an appropriate type combinator.

```
newtype Haf h a f = Haf{unHaf :: h (f a)}
```

```
func x       = Equal (subst Haf unHaf x)
```

Using *func*, it is easy to define a rewrite function that applies a substitution the one but last argument of a type constructor:

```
rewrite'      :: Equal a b
             → Equal c (f a d)
             → Equal c (f b d)
rewrite' x y  = trans y (func (arg x))
```

A rewrite function for the *n* but last argument of a type constructor can be obtained by applying *func* *n* times, as expressed in the following generic definition:

```
rewrite_n x y = trans y (fun_n (arg x))
```

For example the rewrite function for the two but last argument is obtained by applying *func* twice:

```
rewrite2      :: Equal a b
             → Equal c (f a d e)
             → Equal c (f b d e)
rewrite2 x y  = trans y (func (func (arg x)))
```

Unfortunately, Haskell does not support polymorphic kinds, making the type of *func* invalid. Hence the generic solution for type constructors of different arities cannot be given, and instead, we must give specialized definitions for type constructors of different kinds. For example the function *rewrite'* for type constructors of kind $* \rightarrow * \rightarrow *$ can be defined as follows.

Firstly, we need a version of the type *Equal* specialized for type constructors of kind $* \rightarrow *$:

```
data Equal' f g = Equal' (forall h.h f -> h g)
                | KindInfo (f ()) (g ())
```

In order to prevent Haskell's kind inferencer to default the kind of *f* and *g* to $*$, a dummy alternative³ is provided, in order to force the inferencer to infer $* \rightarrow *$ instead.

In a similar way the composition type constructor is specialized, the dummy alternative enforces that *g* has kind $* \rightarrow * \rightarrow *$:

```
data Comp' f g x = Comp' {unComp' :: f (g x)}
                 | KindInfo2 (g () ())
```

³The latest release of GHC supports kind annotations. These provide a more elegant solution for assigning an explicit kind to a type variable, than the use of dummy constructors.

Using these specialized data types, specialized versions of the functions *arg* and *func* can be defined in a similar way as before:

$$\begin{aligned} \text{arg}' & & :: & \text{Equal } a \ b \rightarrow \text{Equal}' (f \ a) (f \ b) \\ \text{arg}' \ ab & & = & \text{Equal}' (\text{subst } \text{Comp}' \ \text{unComp}' \ ab) \\ \\ \text{subst}' & & :: & (ta \rightarrow c \ a) \\ & & \rightarrow & (c \ b \rightarrow tb) \\ & & \rightarrow & \text{Equal}' \ a \ b \\ & & \rightarrow & ta \\ & & \rightarrow & tb \\ \text{subst}' \ \text{from } to \ (\text{Equal}' \ ab) & = & to.ab.\text{from} \\ \\ \text{func}' & & :: & \text{Equal}' \ f \ g \rightarrow \text{Equal} (f \ a) (g \ a) \\ \text{func}' \ fg & & = & \text{Equal} (\text{subst}' \ \text{Haf} \ \text{unHaf} \ fg) \end{aligned}$$

Finally, a valid definition for *rewrite'* can be given:

$$\begin{aligned} \text{rewrite}' & & :: & \text{Equal } b \ d \\ & & \rightarrow & \text{Equal } a \ (f \ b \ c) \\ & & \rightarrow & \text{Equal } a \ (f \ d \ c) \\ \text{rewrite}' \ a \ b & & = & \text{trans } b \ (\text{func}' \ (\text{arg}' \ a)) \end{aligned}$$

Fortunately, specializations for more complex kinds are not required, as they are not used in the remainder of this paper.

Using the combinators defined above, the law of congruence for a binary functor:

$$\text{CONGRUENCE: } \frac{a \equiv b, c \equiv d}{f \ a \ c \equiv f \ b \ d}$$

can be defined as follows:

$$\begin{aligned} \text{congruence} & & :: & \text{Equal } a \ b \\ & & \rightarrow & \text{Equal } c \ d \\ & & \rightarrow & \text{Equal} (f \ a \ c) (f \ b \ d) \\ \text{congruence } ab \ cd & = & \text{rewrite } cd \ (\text{rewrite}' \ ab \ \text{reflex}) \end{aligned}$$

Equivalence proofs can be constructed easily using the proof combinators defined thus far. Consider for example the following proposition and its proof:

$$x \equiv a \rightarrow b, y \equiv c \rightarrow d, a \equiv c, b \equiv d \vdash x \equiv y$$

- 1 $x \equiv a \rightarrow b$ (assumption)
- 2 $y \equiv c \rightarrow d$ (assumption)
- 3 $a \equiv c$ (assumption)
- 4 $b \equiv d$ (assumption)
- 5 $a \rightarrow b \equiv c \rightarrow d$ (congruence: 3, 4)
- 6 $c \rightarrow d \equiv y$ (symmetry: 2)
- 7 $x \equiv c \rightarrow d$ (transitivity: 1,5)
- 8 $x \equiv y$ (transitivity: 7,6)

This proof is easily encoded using proof combinators as follows:

$$\begin{aligned} \text{deduce} & & :: & \text{Equal } x \ (a \rightarrow b) \\ & & \rightarrow & \text{Equal } y \ (c \rightarrow d) \\ & & \rightarrow & \text{Equal } a \ c \\ & & \rightarrow & \text{Equal } b \ d \\ & & \rightarrow & \text{Equal } x \ y \\ \text{deduce } x1 \ x2 \ x3 \ x4 & = & \text{let } x5 = \text{congruence } x3 \ x4 \\ & & & x6 = \text{symm } x2 \\ & & & x7 = \text{trans } x1 \ x5 \\ & & & x8 = \text{trans } x7 \ x6 \\ & & \text{in } & x8 \end{aligned}$$

5 Type Representations

Thus far, we have defined an encoding of type equivalence proofs and a set of combinators to construct such proofs. Since we do not want to fix the type used for type representations, we introduce the class *TypeDescr*, that only contains the function (\sim) that is used for checking whether a type representation *tpr a* represents the same type as a type representation *tpr b*. The *Maybe* data type is used to distinguish between a successful comparison, in which case a witness of type *Equal a b* is returned, and an unsuccessful one, in which case *Nothing* is returned.

$$\begin{aligned} \text{data } \text{Maybe } a & & = & \text{Just } a \\ & & | & \text{Nothing} \end{aligned}$$

$$\begin{aligned} \text{class } \text{TypeRep } tpr \ \text{where} \\ (\sim) & :: tpr \ a \rightarrow tpr \ b \rightarrow \text{Maybe} (\text{Equal } a \ b) \end{aligned}$$

We continue by defining the function *coerce* for the *Equal* type, using the function *subst* and the identity type constructor:

$$\begin{aligned} \text{newtype } \text{Id } x & & = & \text{Id}\{\text{unId} :: x\} \\ \\ \text{coerce} & & :: & \text{Equal } a \ b \rightarrow (a \rightarrow b) \\ \text{coerce } ab & & = & \text{subst } \text{Id } \text{unId } ab \end{aligned}$$

Using the operator (\sim) the function *fromDyn*, that converts a dynamic into a normal value, can now easily be expressed:

$$\begin{aligned} \text{fromDyn} & & :: & \text{TypeRep } tpr \\ & & \Rightarrow & tpr \ a \rightarrow \text{Dynamic } tpr \rightarrow \text{Maybe } a \\ \text{fromDyn } et \ (x :: t) & = & \text{case } t \sim et \ \text{of} \\ & & & \text{Just } eq \rightarrow \text{Just } (\text{coerce } eq \ x) \\ & & & \text{Nothing} \rightarrow \text{Nothing} \end{aligned}$$

The function *fromDyn* checks whether the expected type *et* matches the type of the value stored in its second argument. If this is the case the stored value is “converted”, using the constructed evidence *eq*, into the expected type and returned. Note that the only way in which this conversion can be done, is by applying the conversion function returned by the call of (\sim) in case the types match.

We continue by defining a data type *TpCon* that represents type constants, such as *Int* and *Bool*. The constructors *Int* and *Bool*, take a proof that the type-label *a* equals the type *Int* respectively *Bool*:

$$\begin{aligned} \text{data } \text{TpCon } a & & = & \text{Int } (\text{Equal } a \ \text{Int}) \\ & & | & \text{Bool } (\text{Equal } a \ \text{Bool}) \end{aligned}$$

Since we want to be able to compare such types we make it an instance of *TypeRep*. The function (\sim) checks whether both its arguments are structurally equivalent and if this is the case constructs a proof that both arguments are of the same type using the transitivity and symmetry combinators.

$$\begin{aligned} \text{instance } \text{TypeRep } \text{TpCon} \ \text{where} \\ (\sim) \ (\text{Int } x) \ (\text{Int } y) & = \text{Just } (\text{trans } x \ (\text{symm } y)) \\ (\sim) \ (\text{Bool } x) \ (\text{Bool } y) & = \text{Just } (\text{trans } x \ (\text{symm } y)) \\ (\sim) \ _ \ _ & = \text{Nothing} \end{aligned}$$

Smart constructors for the type representations of the types *Int* and *Bool* are defined as *inttp* and *booltp*:

$$\begin{aligned} \text{inttp} & & :: & \text{TpCon } \text{Int} \\ \text{inttp} & & = & \text{Int } \text{reflex} \\ \text{booltp} & & :: & \text{TpCon } \text{Bool} \\ \text{booltp} & & = & \text{Bool } \text{reflex} \end{aligned}$$

Using these we can easily inject values of type *Int* and *Bool* together with their type representations into a *Dynamic*.

```
true           = True :: booltp
ninetythree    = 93  :: inttp
```

Now the operation *fromDyn* can be used to try to convert dynamic values into a concrete type. For example: *fromDyn booltp true* results in *Just True* and *fromDyn booltp ninetythree* will fail.

Since we now not only want to deal with constant types we introduce a new type *TpRep* that extends the type representation with alternatives for list and function types, containing the representations of the constituting types:

```
data TpRep tpr a = TpCon (tpr a)
                 | ∃x.List (Equal a [x])
                   (TpRep tpr x)
                 | ∃x y.Func (Equal a (x → y))
                   (TpRep tpr x)
                   (TpRep tpr y)
```

For the new type descriptor *TpRep* the smart constructors of the representations of *Int* and *Bool* need to be redefined:

```
type Type       = TpRep TpCon

inttp           :: Type Int
inttp           = TpCon (Int reflex)
booltp         :: Type Bool
booltp         = TpCon (Bool reflex)
```

The smart constructor *list* for list types takes as an argument the representation of the component type:

```
list           :: TpRep tpr a → TpRep tpr [a]
list tpr_a    = List reflex tpr_a
```

For function types the right-associative operator *.→.* is defined, that constructs a function type representation out of its argument and result type representation.

```
(.→.)         :: TpRep tpr a
              → TpRep tpr b
              → TpRep tpr (a → b)
a .→. r       = Func reflex a r
```

The smart constructors provide a convenient notation for constructing type representations; for example the representation for *[Int] → Bool* can be expressed as follows: *list inttp .→. booltp*.

By now making *TpRep* an instance of the class *TypeDescr* it becomes possible to cast function types and list types. In order to match two type constants the function (*~*) on the underlying representation of type constants is called.

```
instance TypeRep tpr ⇒ TypeRep (TpRep tpr) where
  (~) (TpCon x) (TpCon y) = x ~ y
```

For list types the component types are compared, resulting in an equivalence proof of both component types if the comparison succeeds. This proof is subsequently used to construct the proof that both list types match.

```
(~) (List x t1) (List y t2) = case t1 ~ t2 of
  Just eq → Just (trans (rewrite eq x) (symm y))
  Nothing → Nothing
```

The definition of (*~*) on two function types is quite straightforward: if we can prove that their argument types are equal and we can prove that their result types are equal then we may deduce that both function types are equal.

```
(~) (Func x a1 r1) (Func y a2 r2) =
  case (a1 ~ a2, r1 ~ r2) of
    (Just arg, Just res) → Just (deduce x y arg res)
    _ → Nothing
```

Finally, if the type representations do not match then no equivalence proof can be constructed, hence *Nothing* is returned.

```
(~) _ _ = Nothing
```

A function value can be cast into a dynamic value, and later on be down-cast to a value of its original type. During a down-cast a type-check is performed once, and afterwards we can use the function many times without again paying for type-checking. Consider for example the function *plus* and the value *one*, resulting from an up-cast of the operator (+) and the integer 1, respectively:

```
plus           :: Dynamic Type
plus          = (+) :: inttp .→. inttp .→. inttp

one            :: Dynamic Type
one           = 1 :: inttp
```

Two values can be applied, if the first one has a function type and the type of other matches the argument type of the function:

```
dynApply       :: TypeRep tp
              ⇒ Dynamic (TpRep tp)
              → Dynamic (TpRep tp)
              → Maybe (Dynamic (TpRep tp))

dynApply (f :: ft) = case ft of
  Func eqf arg res →
    let f' = coerce eqf f
        in λ(x :: xt) → case xt ~ arg of
          Just eqa → let x' = coerce eqa x
                    in Just (f' x' :: res)
          Nothing → Nothing
  _ → const Nothing -- not a function type
```

The value *inc* is the result of the application of *plus* to *one*:

```
inc           :: Dynamic Type
inc          = case dynApply plus one of
              Just v → v
```

The value *inc* contains the function ((+) 1) together with a description of its type, which is *(Int → Int)*. The function *increment* is the result of a down-cast of *inc*. After the down-cast we end up with a function that adds 1 to a number without performing the type-check again.

```
increment     :: Int → Int
increment     = case fromDyn (inttp .→. inttp) inc of
              Just f → f
```

We want to stress once more that inside the expressions that are constructed there initially live many applications of identity functions to values. When our function is evaluated for the first time, all such applications of identities will be removed, and the overhead resulting from type checking the expression when building it is gone.

6 Interpreting Expressions

In this section we show how to use the machinery from the previous section in constructing efficient typed evaluators. We do so in the context of a syntax macro mechanism, in which an already existing parser for some language is dynamically extended to recognize and process a larger language. A lot of research has been done on syntax macros and related topics[7, 10, 3, 14]. Our notation for syntax macros is borrowed from Cardelli *et al.*[3].

A syntax macro enabled compiler first reads a set of macro definitions –and after having processed them successfully– reads the actual program in the extended language. The semantics associated with the new language constructs is expressed in terms of expressions of a base language, and are in our case built from constructors describing the abstract syntax of the base language.

Since these definitions are parsed, processed and evaluated by the existing compiler we want to make sure that after successfully processing the macro definitions, we do not pay (much) more than when we had added the processing of the new constructs directly to the existing compiler. Furthermore we want to make sure that the compiler, after having successfully processed the macro definitions, does not break down due to a typing error when processing the actual program.

As an example we take a very simple language with the following abstract syntax:

```

data Exp      = Add Exp Exp
                | Sub Exp Exp
                | IntLit Int

```

A set of syntax macros that defines a concrete syntax for this language typically looks as follows:

```

Expr, Factor :: Exp
FactIter     :: Exp -> Exp
Oper         :: Exp -> Exp -> Exp

Expr ::= x=Factor
      fs=FactIter => fs x
      | x=Factor  => x

FactIter ::= op=Oper y=Factor
          fs=FactIter => \x:Exp . fs (op x y)
          |             => \x:Exp . x

Oper ::= "+" => Add
      | "-" => Sub

Factor ::= x=IntLiteral => IntLit x

```

The macros start with the declaration of the types of new concrete nonterminals that are introduced by the macros. The constants in this type language are the nonterminals from the abstract syntax.

The declarations are followed by a list of production rules. The part of a production rule left of the arrow resembles a BNF production and defines the concrete syntax of the language. The right-hand side of a production rule defines a mapping to the abstract syntax of the core language. The parser constructed for the above macros recognizes expressions containing left associative +, – operators and integer literals, and constructs a parse tree of type *Exp*.

Our syntax macro system constructs parsers at run-time using self-optimizing parser combinators[15] and calls this parser directly on a source file. It is not necessary to generate parse-tables off-line to improve efficiency as is common in most other implementations.

The idea behind combinator parsers is that a parser is a function that consumes a list of symbols and produces an abstract syntax tree of some type. The problem that we have to deal with now is that parsers must be well-typed just as any other function. Thus the macro system interprets a macro and has to construct a well-typed parser. In order to achieve this, run-time type-checking is required. Using the type-checking approach introduced before, well-typed parsers can be constructed in such a way that the cost of type-checking is paid for only once.

To illustrate the use of run-time type-checking during interpretation of the macros we show how the interpretation of the right-hand sides of the macro is done.

The abstract syntax of the expressions at the right-hand side of a production rule is defined by the following data type:

```

type Ident    = String
data Expression = Var Ident
                  | Apply Expression Expression
                  | Const (Dynamic Type)
                  |  $\exists x.$ Lambda Ident (Type x) Expression
                  |  $\exists x.$ Let Ident (Type x)
                      Expression
                      Expression

```

Underlying the type checking is the concept of type judgments:

$$\Gamma \vdash \text{expr} : t$$

We read this as follows: under the assumptions about the types of the variables in Γ , the expression *expr* is well typed and has type *t*.

The typing rule for constant values, like 3 and *True*, is trivial. It states that a constant of type *tp* has type *tp*:

$$\text{CONSTANT} : \Gamma \vdash c_{tp} : tp$$

Typing an expression consisting of a single identifier is described by:

$$\text{IDENT} : \frac{(id : t) \in \Gamma}{\Gamma \vdash id : t}$$

The rule for function application is:

$$\text{APPLY} : \frac{\Gamma \vdash \text{expr}_1 : a \rightarrow r, \Gamma \vdash \text{expr}_2 : a}{\Gamma \vdash \text{expr}_1 \text{expr}_2 : r}$$

The rule for lambda abstraction is:

$$\text{LAMBDA} : \frac{(\Gamma - id) \cup \{id : a\} \vdash \text{expr} : b}{\Gamma \vdash (\lambda id : a . \text{expr}) : a \rightarrow b}$$

Finally, the rule for let expressions is:

$$\text{LET} : \frac{(\Gamma - id) \cup \{id : a\} \vdash \text{expr}_1 : a, (\Gamma - id) \cup \{id : a\} \vdash \text{expr}_2 : b}{\Gamma \vdash (\text{let } id : a = \text{expr}_1 \text{ in } \text{expr}_2) : b}$$

In our compiler we distinguish a compile-time and a run-time environment. The compile-time environment is used during type-

checking and compilation, it is a symbol table containing the type and the location in the run-time environment for each variable. The run-time environment is used during evaluation of an expression and contains the actual values for the variables in the expression.

Nested pairs are a very suitable data structure for the run-time environment. They can store heterogenous data, which is important because the variables in an expression may have different types. Further more selector functions to access each value can easily be constructed using the function *fst* and *snd*. Consider for example an environment containing the values *True*, 3, and 'a':

$(True, (3, 'a'))$

The selector functions for the first, second and third value are respectively *fst*, *fst.snd*, and *snd.snd*.

Compiled expressions need a run-time environment to provide values for their variables. Hence a compiled expression is represented as a function of type $(env \rightarrow a)$, that computes the value of the expression of type *a*, given an environment of type *env*, containing a valuation for its variables. As the result type of a compiled expression is dynamic, the function representing the expression is wrapped as a dynamic value.

The data type *DynamicF* is a generalization of *Dynamic*.

data *DynamicF* *tpr f* = $\exists x.f\ x :: tpr\ x$

The functions for converting dynamic values into normal values can be defined for the type *DynamicF* in a similar way as done for the type *Dynamic*:

coerceF :: *Equal a b* $\rightarrow f\ a \rightarrow f\ b$
coerceF (*Equal eq*) = *eq*

fromDynF :: *TypeRep tpr*
 $\Rightarrow tpr\ a$
 $\rightarrow DynamicF\ tpr\ f$
 $\rightarrow Maybe\ (f\ a)$

fromDynF e (*x* :: *t*) = **case** *t* ~ *e* **of**
Just eq $\rightarrow Just\ (coerceF\ eq\ x)$
Nothing $\rightarrow Nothing$

The type of compiled expressions is defined by parametrizing *DynamicF* with the type constructor $((\rightarrow) env)$:

type *CompExp env* = *DynamicF Type* $((\rightarrow) env)$

The compile-time environment is a symbol table storing for each variable its type and a the location of the variable at run-time. The type of an identifier is encoded as a type representation of type *Type t*, and the location as a selector function of type $(env \rightarrow t)$. The identifier information consisting of a type representation and a selector function are packed as a dynamic value:

type *IdentInfo env* = *DynamicF Type* $((\rightarrow) env)$

The type of the symbol table is defined as:

type *SymTable env* = $[(Ident, IdentInfo\ env)]$

Each time a declaration of variable is introduced, a new entry with the type and selector function of the variable must be added to the symbol table. Furthermore, the type of the run-time environment has to change, since it has to store a value for the new identifier. The function *add* takes a pair of an identifier and the representation of

its type and extends the environment with this new identifier.

add :: $(Ident, Type\ a)$
 $\rightarrow SymTable\ env$
 $\rightarrow SymTable\ (a, env)$
add (*x, tp*) *gamma* = $(x, fst :: tp) : map\ f\ gamma$
where *f* (*ident, get* :: *t*) = $(ident, (get.snd) :: t)$

In order to hold a value of type *a* for the new identifier, the type of the run-time environment is changed to a pair of a value of type *a* and the old environment. The selector function for the new identifier is obviously the function *fst*. As the type of the run-time environment changed from *env* to (a, env) , the selector functions for the other identifiers must be composed with *snd*.

We proceed by developing a compiler for the simple expression language defined above. As the structure of the compiler function closely follows the typing rules, we choose it to resemble the type judgement operator:

$(|-) :: SymTable\ env \rightarrow Expression \rightarrow Maybe\ (CompExp\ env)$

This operator takes an environment and an expression as arguments; if all variables occurring in the expressions are introduced in the environment and there are no type errors then the compilation succeeds and returns the value of the expression as a compiled expression. The monadic **do** notation is used in the definition of $(|-)$ to deal with the *Maybe* type. It avoids cluttering the code with pattern matches on *Nothing* and *Just* and makes the resemblance of the implementation to the typing rules more clear.

The definition of the compile operator for constants is straightforward. No environment is needed to evaluate a constant, so the environment is discarded by the function *const* and the value of the constant is returned.

gamma $|-$ *Const* (*c* :: *tp*) = *return* (*const c* :: *tp*)

The interpretation of function applications closely follows the corresponding typing rule. The first three lines of the **do** statement correspond to the three assumptions in the typing rule, whereas the last line builds the conclusion. The coerce functions *funEQ* and *argEQ* are applied in order to convince the type-checker that *e1* is a function and the type of *e2* matches the argument type of this function. In the resulting expression the environment *env* is passed to both expressions.

gamma $|-$ *Apply* *expr1* *expr2* =
do *e1* :: *Func funEQ a r* $\leftarrow gamma\ |- expr1$
e2 :: *b* $\leftarrow gamma\ |- expr2$
argEQ $\leftarrow b \sim a$
let *e1'* = *coerceF funEQ e1*
e2' = *coerceF argEQ e2*
return $((\lambda env \rightarrow e1'\ env\ (e2'\ env)) :: r)$

For the interpretation of variables, the identifier is located in the environment *gamma*, yielding the representation of its type and a function that selects its value from the run-time environment:

gamma $|-$ *Var* *x* = **do** *var* :: *tp* $\leftarrow lookup\ x\ gamma$
return (*var* :: *tp*)

The interpretation operation for lambda-abstractions adds the identifier with its type to the symbol table and uses the extended symbol table to compile the body of the lambda. At run-time the value for the identifier is placed into the run-time environment, which is subsequently used to compute the value of the body.

```

gamma|-Lambda x tp expr =
  do e ::: etp      ← add (x,tp) gamma|-expr
  let mkLam env =  λx → e (x,env)
  return (mkLam ::: tp .→. etp)

```

The compilation of a let expression proceeds in a similar way. During compilation the symbol table is extended with the identifier and type of the declaration. This symbol table is then used to compile both the declaration and the body. At run-time the value of the declaration is put into the environment, which is used to evaluate both the declaration and the body of the let expression.

```

gamma|-Let x tp expr1 expr2 =
  do let gamma' = add (x,tp) gamma
  e1 ::: tp1 ← gamma'|-expr1
  e2 ::: tp2 ← gamma'|-expr2
  eq        ← tp1 ~ tp
  let mkLet env = let env' = (decl,env)
                  decl = coerceF eq e1 env'
                  in e2 env'
  return (mkLet ::: tp2)

```

Using the compiling operator we can compile expression terms into their values:

```

compile      :: Expression → Type a → Maybe a
compile expr t = do res ← []|-expr
                val ← fromDynF t res
                return (val ())

```

The function *compile* takes an expression term and starts with the empty environment (`[]`). If the resulting value matches the expected type an empty run-time (`()`) environment is supplied, and the value of the expression is returned.

As a final example now consider the expression corresponding to the lambda-term $\lambda x :: Int. \lambda y :: Int. x + y$:

```

expr      :: Expression
expr      = (Lambda "x" inttp
            (Apply
              (Apply
                (Const plus)
                (Var "x")
              )
              (Var "y")
            )
          )

```

This term is compiled into a function with type $Int \rightarrow Int \rightarrow Int$ as follows:

```

test :: Int → Int → Int
test = case compile expr (inttp .→. inttp .→. inttp) of
        Nothing → error "type_error_in_expression"
        Just x → x

```

The function *test* can now be applied to add two integers, without any further type checking being involved!

The expression compiler can also deal with recursive definitions as illustrated by the following example. Consider the following expression containing a recursive definition.

```
let ones = 1 : ones in ones
```

This expression is encoded as a term (*expr2*) of type *Expression*, using the helper functions *cons* and *int* to wrap the operator (`:`) and the integer value, respectively.

```

cons      :: Type t → Expression
cons t    = Const ((: ::: t .→. list t .→. list t)

int       :: Int → Expression
int x     = Const (x ::: inttp)

expr2     :: Expression
expr2     = Let "ones" (list inttp)
              (Apply (Apply (cons inttp) (int 1))
                    (Var "ones"))
              (Var "ones")

```

The expression *expr2* can be compiled to a Haskell value of type `[Int]` as follows:

```

test2 :: [Int]
test2 = case compile expr2 (list inttp) of
        Nothing → error "type_error_in_expression"
        Just x → x

```

7 Conclusions

We have shown that extending a language with a separate dynamic typing mechanism is not needed, provided the core language is sufficiently rich. The techniques presented here can easily be incorporated within a small module. Our solution does not require large compiler modifications or ad hoc language extensions such as *unsafeCoerce*. The presented approach is type safe and as powerful as Haskell's dynamic typing library. In contrast to some of the other approaches to dynamic typing, our library does not support polymorphic dynamic values. Whether our approach can easily be extended with dynamic polymorphism is as yet unknown and a subject of further research.

In order to use the dynamic typing library an instance of the class of type representations must be provided. The code for these instances is completely generic and can be easily generated by a special tool; or by an extended version of the Generic Haskell compiler [4, 5]. Another solution is to let the compiler construct the type representations as an abstract data type. This opens the way for dynamic linking and reflection, e.g. by exporting from each module its environment in the form of a function of the type:

```
environment :: FiniteMap String Dynamic
```

In this way run-time access to the module's interface (`.hi`-file) is obtained. Occurrences of global variables in the expression terms that are compiled by the expression interpreter developed in the previous section could in that case be linked to the corresponding functions in the environment.

We have introduced a data type *Equal* to represent genuine type equality. It finds its theoretical basis in Leibnitz' definition of equality and the Curry-Howard isomorphism. We have developed a small library of proof combinators to constructs of the *Equal* type. Values of the type *Equal* are represented by identity functions. By combining these with function application or composition, the proof combinators construct new values of the *Equal* type. The proof combinators correspond to proof rules and make up a small proof system.

A dynamic value is cast to a concrete type by comparing the type of the dynamic value with the expected type, resulting in a proof that they are equal if they are. This proof is a composition of identity functions. The identity functions are applied to the dynamic value, coercing it to a value of the expected type. After the dynamic type checking overhead has been computed once, the resulting value is free of typing overhead and can subsequently be used just as a normal value.

References

- [1] 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, April 1991.
- [2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Summary in *ACM SIGPLAN Workshop on ML and its Applications*, June 1992.
- [3] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Systems Research Center, 1994.
- [4] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löf, and Jan de Wit. The Generic Haskell user’s guide. Technical Report UU-CS-2001-26, Institute of Information and Computing Sciences, Utrecht University, 2001.
- [5] Dave Clarke and Andres Löf. Generic Haskell, specifically. In *IFIP WG2.1 Working Conference on Generic Programming*, 2002.
- [6] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994. Selected papers of the Fourth European Symposium on Programming (Rennes, 1992).
- [7] B. M. Leavenworth. Syntax macros and extended translation. *CACM*, 9(11):790–793, 1966.
- [8] Xavier Leroy and Michael Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, volume 523, pages 406–426, Berlin, Heidelberg, New York, 1991. Springer-Verlag.
- [9] Simon Peyton-Jones, John Hughes, and (eds). Report on the programming language Haskell 98. <http://www.haskell.org/report>, February 1998.
- [10] Simon L. Peyton Jones. Parsing distfix operators. *CACM*, 29:118–122, 1986.
- [11] Marco Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages, 10th International Workshop, IFL’98*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer, 1999.
- [12] Rinus Plasmeijer and Marco van Eekelen. Clean language report version 2.0. <http://www.cs.kun.nl/clean/>, 2002.
- [13] Mark Shields, Tim Sheard, and Simon L. Peyton Jones. Dynamic typing as staged type inference. In *Symposium on Principles of Programming Languages*, pages 289–302, 1998.
- [14] Guy L. Steele Jr. Growing a language. *Journal of Higher-Order and Symbolic Computation*, 12:221–236, 1999.
- [15] Doaitse Swierstra. Combinator parsers: From toys to tools. In Graham Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.
- [16] Stephanie Weirich. Type-safe cast. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 58–67. ACM Press, 2000.
- [17] Zhe Yang. Encoding types in ML-like languages. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 289–300. ACM Press, 1998.