Saint: an API-generic Type-safe Interpreter*

Maximilian Algehed[1 orcid], Patrik Jansson[1 orcid],
Sólrún Halla Einarsdóttir[1 orcid], and Alex Gerdes[1,2 orcid]

[1] Chalmers University of Technology, Gothenburg, Sweden,
{algehed,patrikj,slrn}@chalmers.se,
[2] University of Gothenburg, Gothenburg, Sweden,
alex.gerdes@cse.gu.se

Abstract. Typed functional programming allows us to write interesting programs without sacrificing type safety. Programs that expose its API to an open world, however, are faced with the problem of dynamic type checking. In Haskell, existing techniques that address this problem, such as `Typeable` and `Dynamic`, are often closed and difficult to extend. We have constructed an extensible Haskell library for describing APIs using *annotated* type representations. As a result, API calls can be interpreted in a type-safe manner without extra programming effort. In addition, the user has full control over the universe of allowed types, which helps to catch misconceptions in an early stage. We have applied our technique to connect a real-world DSL (GRACe) to a JavaScript GUI.

Keywords: Domain Specific Language · Interpreter · Lambda Calculus

1 Introduction

A large number of so-called Embedded Domain Specific Languages (EDSLs) have been implemented in Haskell for various purposes [16,1,7,11]. Embedding DSLs in a typed language like Haskell has many advantages, one of the major ones being that it removes the need for the implementation of tools like parsers and type-checkers. However, embedded languages require a full Haskell environment in order to be compiled and executed.

A Haskell EDSL may be used as part of a larger toolchain, in combination with other programs that may not be written in Haskell and may even be written in an untyped language. This necessitates communication between the Haskell EDSL and the untyped world. For instance, we may want to expose functions from an EDSL as an API to an untyped frontend, accessible through a web interface, and allow users to write programs using those functions which can be sent to our Haskell backend for execution.

For example, consider a small Haskell EDSL to build pictures using a set of functional geometry (FunGeo) combinators, as described by Henderson in [8,9]. An overview of the system can be seen in Figure 1. The FunGeo EDSL consists

* Post-print of paper from TFP 2018, LLNCS version to appear, published by Springer.

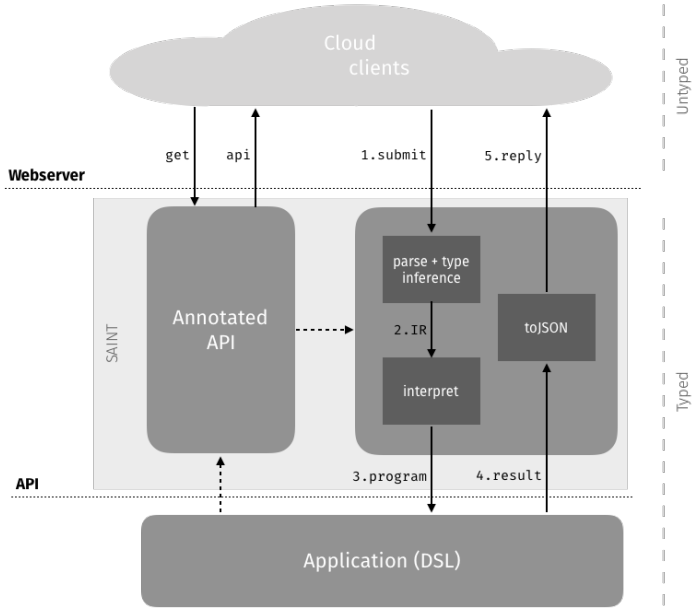


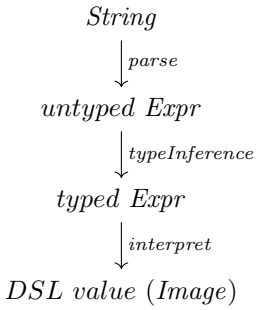
Fig. 1. A high-level overview of the system

of a datatype for images and some image combinators. We have added *natrec* to make it a bit more powerful and to make sure more than one type is involved.

```
beside :: Image → Image → Image
above  :: Image → Image → Image
over   :: Image → Image → Image        -- overlay
rot    :: Image → Image                -- 90 degrees
fish   :: Image                        -- a simple fish to start from
data Image                             -- implemented as just a list of splines
natrec :: Image →                      -- base case
          (Int → Image → Image) →      -- step function
          Int → Image                  -- main input and output
```

We would like to expose the ability to program safely in FunGeo where "safe" means "type correct"; we want to rule out raw expressions like *beside* 3 *fish* or *fish fish*. We do this in the following manner:

1. Parse the user-written program into an untyped expression (and find syntax errors).
2. Infer the types of the expression (find type errors).
3. Interpret the typed expression as a DSL value.
4. Send the result of the DSL computation back.



An example program and its output is shown in Fig. 2.

```
let fish2 = flip (rot45 fish) in
let fish3 = rot (rot (rot fish2)) in
let t     = over fish (over fish2 fish3) in
let u     = over (over fish2 (rot fish2))
                 (over (rot (rot fish2))
                       fish3) in
let qrt   = \p.\q. \r.\s.
              above  (beside p q)
                     (beside r s) in
let cyc   = \p. qrt p (rot p)
                      (rot (rot p))
                      (rot (rot (rot p))) in
let side  = natrec blank (\n.\img.
               qrt img     img
                   (rot t)  t)  in
let corn  = natrec blank (\n.\img.
               qrt img            (side n)
                   (rot (side n)) u )  in
let bes3  = \a.\b.\c. besideS 1 2 a (beside b c) in
let abo3  = \a.\b.\c. aboveS  1 2 a (above  b c) in
let nnet  = \p.\q.\r. \s.\t.\u. \v.\w.\x. abo3 (bes3 p q r)
                                               (bes3 s t u)
                                               (bes3 v w x) in
let sqrl  = \n. nnet (corn n)       (side n) (rot (rot (rot (corn n))))
                     (rot (side n)) u        (rot (rot (rot (side n))))
                     (rot (corn n)) (rot (rot (side n))) (rot (rot (corn n))) in
scale 1000 (sqrl 3)
```

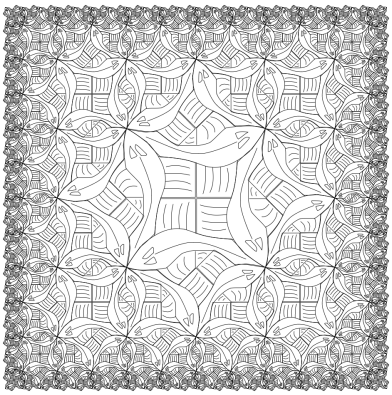


Fig. 2. The Escher Woodcut — Square Limit — source code and output.

1.1 APIs as values

In order to expose our API of DSL functions to users, we describe it as a value of the type *Library*, which is a type-annotated lookup table containing an *Item* describing each function. An *Item* contains a function's name, its semantics, and a representation of its type. The name can be used for parsing, the semantics for interpretation, and the type representation for type checking. Our running example, the FunGeo EDSL, is easily described as a *Library* value.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Library} = \mathit{Library}\ \mathit{String}\ [\mathit{Item}]\\
&\textbf{data}\ \mathit{Item}\ \ \ = \mathit{Item}\ \mathit{String}\ \mathit{TypedValue}\\
&\mathit{funGeoLib} :: \mathit{Library}\\
&\mathit{funGeoLib} = \mathit{Library}\ \texttt{"funGeo"}\ (\mathit{funGeoCore} +\!\!+ \mathit{funGeoMore})\\
&\mathit{funGeoCore}, \mathit{funGeoMore} :: [\mathit{Item}]\\
&\mathit{funGeoCore} =\\
&\quad [\mathit{Item}\ \texttt{"beside"}\ \$\ \mathit{beside} ::: \mathit{image} \dashrightarrow \mathit{image} \dashrightarrow \mathit{image}\\
&\quad , \mathit{Item}\ \texttt{"above"}\ \ \$\ \mathit{above} ::: \mathit{image} \dashrightarrow \mathit{image} \dashrightarrow \mathit{image}\\
&\quad , \mathit{Item}\ \texttt{"over"}\ \ \ \$\ \mathit{over}\ \ ::: \mathit{image} \dashrightarrow \mathit{image} \dashrightarrow \mathit{image}\\
&\quad , \mathit{Item}\ \texttt{"rot"}\ \ \ \ \$\ \mathit{rot}\ \ \ \ ::: \mathit{image} \dashrightarrow \mathit{image}]
\end{aligned}
$$

Here, triple colon (:::) is used to pair a semantic value with a representation of its type. The type representations are built from the base type representations (*image*, *int*), the function type combinator ($\dashrightarrow$), and $\mathit{Tag} = (\#) :: \mathit{String} \to \mathit{TypeRep} \to \mathit{TypeRep}$. Our *TypeRep* is explained, and extended, in Section 2.

We use tags to annotate parts of the API with appropriate metadata, which can be used to display the API in the frontend.

```
funGeoMore = [ Item "natrec" $
                   natrec ::: "Recursion over Nat" #
                     image -->
                     "Step function" # (int --> image --> image) -->
                     int --> image
             , Item "fish"    $ fish ::: "Base case" # image ]
```

1.2 Type representations

At the core of the *Library* datatype is the type *TypedValue* which stores a value and a representation of its type. A first implementation version could be this:

```
data TypedValue where (:::) :: a → TypeRep a → TypedValue
```

Note that the type a is existentially quantified, which means that we can store values of different types in, say, a list of typed values, *ex1* :: [*TypedValue*]:

```
ex1 = [((1+) :: Int → Int) ::: int --> int, (3 :: Int) ::: int]
```

The representation of types, *TypeRep* is also parameterised by an a, the role of this parameter is to "keep track" of the type a *TypeRep* represents:

```
data TypeRep a where
  TRImage :: TypeRep Image
  TRInt   :: TypeRep Int
  TRFun   :: TypeRep a → TypeRep b → TypeRep (a → b)
  TRList  :: TypeRep a → TypeRep [a]
image = TRImage; int = TRInt; (-->) = TRFun
infixr 1 -->
```

The presented type representation and typed values are closely related to the Haskell library *Dynamic* together with its *TypeRep* type family [17]. We found that *Dynamic* and *TypeRep* almost, but not quite, provide the functionality we need. Their *TypeRep* is "deep" in that it represents the full type, and support for it is also built-in to GHC. However, their technique is rigid: *TypeRep* cannot be extended with extra constructs. We extend it to store tags *in the type (representation) tree* by introducing a variant with the *Tag* constructor.

```
data TypeRep t where
  TInt  :: TypeRep Int
  TFun :: TypeRep a → TypeRep b → TypeRep (a → b)
  Tag   :: String → TypeRep a → TypeRep a
int = TInt; (-->) = TFun; (#) = Tag
```

Usually this kind of type family is used to represent singleton types, where each (type) family member *TypeRep t* contains just one proper value $tr :: \mathit{TypeRep}\ t$ which is the representation of t. In our case, due to the *Tag*s, there are more variants possible. It would be possible to represent these tags at the type level, but we wanted to keep the library reasonably simple.

1.3 Towards a type-safe interpreter

We can define a generic parser to a "raw" syntax tree (using a datatype for untyped expressions). Using our *Library* as a parameter we then infer types and annotate the syntax tree provided to us by the parser to make typed expressions. In both of these two phases we discard "bad" inputs to make sure the interpreter only receives well-typed expressions to evaluate.

The combination of these phases is what we call a "type-safe interpreter": you can throw any input term at it but only the (syntax- and) type-correct inputs are run. The interpreter function itself still uses an *Either* type to report errors (in case of the *Env* does not cover all variables used, for example) but this should always succeed when called in combination with the type checker.

$$
\begin{aligned}
&\mathit{parse} &&:: \mathit{String} \to \mathit{Maybe}\ \mathit{UExpr} \\
&\mathit{typeCheck} &&:: \mathit{Library} \to \mathit{UExpr} \to \mathit{Maybe}\ \mathit{Expr} \\
&\mathit{interpret} &&:: \mathit{Env} \to \mathit{Expr} \to \mathit{Maybe}\ \mathit{TypedValue} \\
&\mathit{libToEnv} &&:: \mathit{Library} \to \mathit{Env} \\
&\mathit{run} &&:: \mathit{Library} \to \mathit{String} \to \mathit{Maybe}\ \mathit{TypedValue}
\end{aligned}
$$

This simplified view is expanded and details are explained in following sections.

1.4 Contributions

In this paper we make the following contributions:

- We construct a framework (called Saint) for exposing a typed API to untyped world (including a parser, type checker, and an interpreter).
- We provide a version of *Typeable* supporting tags (annotations in the *TypeRep*).
- We implement a method of evaluating programs in our framework in a type-safe way as internal Haskell values.
- We present case studies which show the application of our techniques to DSLs used in real-world applications.

All the code for our framework, Saint, and the FunGeo case study can be found online[3]. The Saint framework satisfies the following criteria:

- It is lightweight: we do not need an entire Haskell compiler to interpret code.
- It is reusable: new functionality can easily be added to a DSL.
- Interpretation of programs is type-safe.
- The results of interpreting client programs are available to the server as Haskell values, even when the denotation is a function.
- Type information and annotations are correctly exposed to the DSL user.

[3] The Saint library: `https://github.com/GRACeFUL-project/Saint`, and the case study: `https://github.com/GRACeFUL-project/SaintCaseStudy`.

2 Typed values

Using our framework we can expose the API of an EDSL to external clients and can safely evaluate programs, expressed in terms of the exported API, to Haskell values. The two main components in our framework are: type reflection (representing types as values) and type-safe dynamic typing. This section shows how we have developed our type reflection implementation. The implementation is inspired by Typeable [14] but our approach has some advantages: it is extensible, general, and remains under the control of the programmer rather than being built in to the compiler.

We continue from the *TypeRep* GADT presented in the introduction, but call it *TRep* for brevity. The basic idea behind this encoding is not new, and variants of it appear for example in Eisenberg and Weirich [6]. We will now set up some infrastructure needed for the "type-safe interpreter": type equality, type representation equality, coercion, and computation with "typed values". The infrastructure consists of the following datatype and functions:

```
data a ≡ b
(?=)   :: TRep a → TRep b → Maybe (a ≡ b)
coerce :: TypedValue → TRep a → Maybe a
app    :: TypedValue → TypedValue → Maybe TypedValue
```

At first we define these functions using the basic *TRep* type representation from Section 1 but we will later base them on refined and generalised *TypeRep*s.

An important feature of Typeable [14] is that we can determine equality between types at runtime, based on type reflection. Like Typeable, we represent equality between types by a GADT — the single constructor *Refl* supplies the evidence that two types are equal. Pattern-matching on *Refl* convinces the compiler that two types are in fact the same as shown in the example *foo* below.

```
data a ≡ b where Refl :: a ≡ a
foo :: (a ≡ Int) → a → Int
foo Refl x = x
```

Having defined a notion of equality between types, we can now implement equality checks between type representations:

```
(?=) :: TRep a → TRep b → Maybe (a ≡ b)
TInt          ?= TInt          = return Refl
Tag _ t       ?= t'            = t ?= t'
t             ?= Tag _ t'      = t ?= t'
TFun t0 t1 ?= TFun t0' t1' = do
  Refl ← t0 ?= t0'
  Refl ← t1 ?= t1'
  return Refl
_ ?= _ = Nothing
```

We can construct a *TypedValue*, as we did before, to hide the type of an expression using existential quantification.

```
data TypedValue where (:::) :: a → TRep a → TypedValue
```

We combine a value of type a with its type representation *TRep* a and can subsequently treat that combination as an untyped value. For example, we can store values of different types in a single list. The type representation in a *TypedValue* can be used to 'escape' from the existential quantification and allows us to retrieve the original (typed) value. The function *coerce* retrieves the value from a *TypedValue* if it matches the given expected type representation.

```
coerce :: TypedValue → TRep a → Maybe a
coerce (a ::: t0) t1 = do
  Refl ← t0 ?= t1
  return a
```

This approach gives us a type-safe version of dynamic typing. In our interpreter we use the type information in a *TypedValue* to apply one *TypedValue* to another, because we can check if the actual values have matching types.

```
app :: TypedValue → TypedValue → Maybe TypedValue
app (f ::: TFun a b) (x ::: a') = do
  Refl ← a ?= a'
  return (f x ::: b)
app _ _ = Nothing
```

2.1 Generalising type representations

A downside of our type representation is that the universe of types, that is the set of types we can represent, is hard-coded in the *TRep* datatype. We suffer from the so-called expression problem [19]: adding more base types (or type constructors) requires changing both the implementation of *TRep* as well as functions working on it, such as (?=). A solution to this problem is to use the 'datatypes à la carte'-method, which represents a datatype as a co-product of its constructors [18].

```
data CoProduct f g a = InL (f a) | InR (g a)
```

The idea is that f and g are type constructors which each represent an individual constructor in the type *TRep* seen previously. For example, we could construct type representations for *Int* and *Bool* as the following datatypes *IntT* and *BoolT*:

```
data IntT a  where IntT  :: IntT Int
data BoolT a where BoolT :: BoolT Bool
```

Using these two base type representations we can construct the universe of type representations that can be either *Int* or *Bool* using *CoProduct*:

```
type MyTRep a = CoProduct IntT BoolT a
```

However, constructing values of type *TRep Int* and *TRep Bool* is quite cumbersome, requiring us to make use of the *InL* and *InR* constructors as well as *BoolT* and *IntT*. Datatypes à la carte solves this problem by allowing us to construct a subtyping typeclass which we can use with *CoProduct*, as shown below:

```
class f :< g where
  inject :: f a → g a
  eject  :: g a → Maybe (f a)
```

The code for the simple instances for (:<) is elided:

```
instance f :< f where    -- ...
instance f :< CoProduct f r where
instance  {-# OVERLAPPABLE #-} f :< r ⇒ f :< CoProduct l r where
```

This formulation of (:<) requires the construction of *CoProduct*s to be right-associated to work correctly, because $f :<$ *CoProduct* (*CoProduct g f*) *h* can not be made to hold by the instances above. It is also not possible to add another instance $f :< l \Rightarrow f :<$ (*CoProduct l r*) as this would overlap with the last instance above. Ideally, we would have a disjunctive instance $f :< l \mid f :< r \Rightarrow f :<$ (*CoProduct l r*), but these are not allowed in GHC (and it is not clear how to resolve such instances). We therefore present the user with a "smart constructor" for *CoProduct* in the form of a type family (:+:).

```
type family f :+: g where
  (CoProduct f g) :+: h = CoProduct f (g :+: h)
  f :+: CoProduct g h   = CoProduct f (g :+: h)
  f :+: g               = CoProduct f g
```

Using this smart constructor we can write generic representations, like *int* and *bool* below, to allow us to construct typed values conveniently.

```
int :: IntT :< tr ⇒ tr Int
int = inject IntT
bool :: BoolT :< tr ⇒ tr Bool
bool = inject BoolT
```

In order to make use of our new open type universe in *TypedValue* we need to alter the type slightly to move from a fixed family *TRep* (of codes for types) to a type parameter:

```
data TypedValue tr where (:::) :: a → tr a → TypedValue tr
```

We can now use construct typed values in an open manner:

```
exI :: IntT :< tr ⇒ TypedValue tr
exI = 42 ::: int
```

We can also use our approach to construct representations for types built from type constructors like *Maybe* and $(\rightarrow)$.

```
data MaybeT tr a where
  MaybeT :: tr a → MaybeT tr (Maybe a)
maybe :: MaybeT tr :< tr ⇒ tr a → tr (Maybe a)
maybe = inject ∘ MaybeT
data FunT tr a where
  FunT :: tr a → tr b → FunT tr (a → b)
(-->) :: FunT tr :< tr ⇒ tr a → tr b → tr (a → b)
a --> b = inject (FunT a b)
```

We can use these representations to construct more interesting *TypedValue*s:

```
exMI :: (IntT :< tr, MaybeT tr :< tr) ⇒ TypedValue tr
exMI = Just 42 ::: maybe int
exFI :: (IntT :< tr, FunT tr :< tr) ⇒ TypedValue tr
exFI = (λx → x + 1) ::: int --> int
```

Note that each of these examples (*exI*, *exMI*, *exFI*) encodes in its type constraint the "minimum requirements" of a universe for them to fit into.

2.2 Type equality for generalised *TypedValue*s

Deciding equality between types is an important part of what makes our *TypedValue*s useful, so we need a way to do so in this generalised setting. What we would prefer, following our previous discussion (in Sect. 2), is a function *coerce* :: *TypedValue tr* → *tr a* → *Maybe a*. Recall that the implementation of the coercion function *coerce* shown previously relied on computing a value *Refl* of type $a \equiv a$ by comparing the type representation in the *TypedValue* with the coerced-to type. In *coerce* we did this using a function (?=) :: *TRep a* → *TRep b* → *Maybe* $(a \equiv b)$. We now need to generalise over our type representation, and the natural way to do this is by using a type class:

```
class TypeEquality tr where
  (?=) :: tr a → tr b → Maybe (a ≡ b)
```

Next we need to make sure that instances of *TypeEquality* are modular in the same way that the construction of type universes is modular. The type equality should be extensible in the same way the type representation is. The first step to achieving this is to make sure that *CoProduct*s can be tested for type equality.

```
instance (TypeEquality f, TypeEquality g) ⇒
         TypeEquality (CoProduct f g) where
  InL a ?= InL b = a ?= b
  InR a ?= InR b = a ?= b
  _     ?= _     = Nothing
```

We also show how to construct the instances for $IntT$ and $MaybeT$:

```
instance TypeEquality IntT where
  IntT ?= IntT = Just Refl
instance TypeEquality tr ⇒ TypeEquality (MaybeT tr) where
  MaybeT a ?= MaybeT b = do
    Refl ← a ?= b
    return Refl
```

Now we can finally define our new and improved version of *coerce*:

```
coerce :: TypeEquality tr ⇒ TypedValue tr → tr a → Maybe a
coerce (v ::: a) a' = do
  Refl ← a ?= a'
  return v
```

2.3 Constructing universes

What we need next is the ability to construct a value $t\ a$ to pass to $(:::)$ and *coerce*. We might be tempted to define a type like **type** $TypeUniverse = IntT :+: MaybeT\ TypeUniverse$ in order that we may create types $TypeUniverse\ (Maybe\ Int)$ and $TypeUniverse\ (Maybe\ (Maybe\ Int))$ to serve as type representations. However, we can't do that as GHC disallows cyclic type synonyms, so instead we create a datatype:

```
newtype Close a = Close ((MaybeT Close :+: IntT) a)
```

The choice of the name *Close* is not an accident as the type represents the closure of the $IntT$ and $MaybeT$ operations for constructing a type universe. Type equality is easily implemented for *Close*:

```
instance TypeEquality Close where
  Close a ?= Close b = a ?= b
```

In order to use our "type formers" *int*, *maybe*, etc to construct values of type *Close* we need instances of $:<$ for each type of interest.

```
instance IntT :< Close where
  inject = Close ∘ inject
  eject (Close t) = eject t
instance MaybeT Close :< Close where
  inject = Close ∘ inject
  eject (Close t) = eject t
```

Defining instances like these for every universe we may want to construct can become quite cumbersome. Based on the fact that *Close* looks a lot like the fix

point of a type-level function, it's tempting to write something along the lines of the following (assuming a slightly more advanced type system than Haskell's):

```
data Close f a = Close (f (Close f) a)
instance t :< f (Close f) ⇒ t :< Close f where
  inject = Close ∘ inject
  eject (Close t) = eject t


type MyUniverse = Close (λc → MaybeT c :+: IntT)   -- not Haskell
```

This fails because GHC does not allow lambda abstraction on the type level. We may then be tempted to use a type synonym instead:

```
type MakeUniverse u = MaybeT u :+: IntT
type MyBadUniverse = Close MakeUniverse
```

However, this fails because *MakeUniverse* is partially applied in the definition of *MyBadUniverse*. It would appear there is no good way out of this mess!

But there is: by adding an extra type parameter like the one to *MaybeT* to all type representations including *CoProduct* we can generalise the definition, and type equality for *IntT tr* is constructed the same way as before.

```
data IntT (tr :: ∗ → ∗) a where
  IntT :: IntT tr Int
data CoProduct f g (tr :: ∗ → ∗) a = InL (f tr a) | InR (g tr a)
instance TypeEquality (IntT tr) where
  IntT ?= IntT = Just Refl
```

Now we can use our definition of *Close* to define universes without the recursive occurrence of *Close*:

```
type MyUniverse = Close (IntT :+: MaybeT)
int :: forall tr. IntT tr :< tr ⇒ tr Int
int = inject (IntT :: IntT tr Int)   -- a generic type code for Int
value :: Maybe Int
value = coerce (42 ::: int) (int :: MyUniverse Int)
```

Note that the use of *coerce* in *value* requires an instance of *TypeEquality* for *MyUniverse* in order to type check. The core to this instance is the instance of *TypeEquality* for our new *CoProduct* type:

```
instance  (  TypeEquality (f tr), TypeEquality (g tr))
          ⇒ TypeEquality (CoProduct f g tr) where
```

Because *tr* is used both in the premise and the conclusion of the instance (much like in the instance for *Close* above) we are forced to use the GHC language

extension `UndecidableInstances`. However, this does not cause any issue as the search for an instance will terminate when it hits *IntT tr* or similar instances which do not need to make use of type equality at *tr* in order to work.

It is possible to generalise the type formers from the previous section even further than we have done so far. Namely, it is possible to abstract the definition of any n-ary type representation. For nullary type formers this is straightforward:

```
data A0 typ (univ :: * → *) a where
  A0 :: A0 typ univ typ
instance TypeEquality (A0 typ univ) where
  A0 ?= A0 = Just Refl
int :: forall u. A0 Int u :< u ⇒ u Int
int = inject (A0 :: A0 Int u Int)
```

Unary and binary type formers can be constructed in the same way: here we show the unary case:

```
data A1 f univ a where
  A1 :: univ a → A1 f univ (f a)
instance TypeEquality univ ⇒ TypeEquality (A1 f univ) where
  A1 t ?= A1 t' = do
    Refl ← t ?= t'
    return Refl
maybe :: A1 Maybe u :< u ⇒ u a → u (Maybe a)
maybe = inject ∘ A1
```

And analogously for the binary case:

```
data A2 f univ a where
  A2 :: univ a → univ b → A2 f univ (f a b)
instance TypeEquality univ ⇒ TypeEquality (A2 f univ) where    -- ...
```

What we have obtained, then, is a general framework in which we can represent any type, and we have done it all without needing to change the GHC compiler.

2.4 Implementing tags

As previously discussed, we use *Tag*s to annotate our EDSL types with metadata. To implement tags for our generalised *TypedValue*s, we want a function like:

```
(#) :: String → Close u a → Close u a
```

How would we implement (#)? One option is to add an external type former:

```
data TagT u a where
  TagT :: String → u a → TagT u a
```

$$
\begin{aligned}
&(\#) :: \mathit{TagT}\ u :< u \Rightarrow \mathit{String} \to u\ a \to u\ a \\
&t \# s = \mathit{inject}\ (\mathit{TagT}\ t\ s)
\end{aligned}
$$

But what instance of *TypeEquality* should we give for *TagT*? That depends on what kind of equality we want to consider. If we want it to be the case that two types are not considered equal unless their tags are equal, we can use:

$$
\mathit{TagT}\ s\ a \mathbin{?\!=} \mathit{TagT}\ s'\ b = \textbf{if}\ s \not\equiv s'\ \textbf{then}\ \mathit{Nothing}\ \textbf{else}\ a \mathbin{?\!=} b
$$

However, if we wish for our tags to be transparent so that *TypeEquality* is independent of tags, that is, $(s \# t) \mathbin{?\!=} t = \mathit{Just}\ \mathit{Refl}$, this is not sufficient. To achieve that we need be able to compare a *Tag* to a constructor which is not a *Tag*. The simplest way of doing so is to not consider *Tag* as a separate type former, but rather introduce a separate notion of metadata, which we achieve by making *Tag* part of *Close*:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Close}\ f\ a = \mathit{Tag}\ \mathit{String}\ (\mathit{Close}\ f\ a) \\
&\qquad\qquad\quad \mid\ \mathit{Close}\ (f\ (\mathit{Close}\ f)\ a) \\
&\textbf{instance}\ \mathit{TypeEquality}\ (t\ (\mathit{Close}\ t)) \Rightarrow \mathit{TypeEquality}\ (\mathit{Close}\ t)\ \textbf{where} \\
&\quad \mathit{Tag}\ \_\ t \mathbin{?\!=} t' \qquad\quad = t \mathbin{?\!=} t' \\
&\quad t \qquad\quad \mathbin{?\!=} \mathit{Tag}\ \_\ t' = t \mathbin{?\!=} t' \\
&\quad \mathit{Close}\ t \mathbin{?\!=} \mathit{Close}\ t' = t \mathbin{?\!=} t' \\
&(\#) :: \mathit{String} \to \mathit{Close}\ t\ a \to \mathit{Close}\ t\ a \\
&(\#) = \mathit{Tag}
\end{aligned}
$$

This tagging infrastructure allows us to attach arbitrary information anywhere in a type representation in the form of a *String*. Naturally, this could be generalised to any type of metadata with more structure than a simple type:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Close}\ f\ \mathit{ann}\ a = \mathit{Tag}\ \mathit{ann}\ (\mathit{Close}\ f\ \mathit{ann}\ a) \\
&\qquad\qquad\qquad\qquad \mid\ \mathit{Close}\ (f\ (\mathit{Close}\ f\ \mathit{ann})\ a)
\end{aligned}
$$

Alongside the appropriate instance of *TypeEquality* and definition of $(\#)$.

This concludes our implementation of type representations. The system is extensible, yet allows for fine grained control over the type universe in question. Types can also be annotated with arbitrary information, this feature is particularly useful when exposing EDSLs to the outside world, where different types of metadata may need to be associated to functions in the EDSL.

2.5 The Saint API

We have presented a number of different encodings of type universes and typed values. It is time we took a step back and review the final API of Saint. First we review the generic constructs which form type representations:

$$
\begin{aligned}
&\textbf{data}\ \mathit{A0} :: (t :: *) \qquad\quad (\mathit{univ} :: * \to *)\ a \\
&\textbf{data}\ \mathit{A1} :: (t :: * \to *) \qquad (\mathit{univ} :: * \to *)\ a
\end{aligned}
$$

```
...
data An :: (t :: * → ... → *) (univ :: * → *) a
A0 ::          A0 t u t
A1 :: u a → A1 t u (t a)
...
An :: u a → u b → ... → u x → An t u (t a b ... x)
```

These type formers assume they are given a universe u. To construct a representation of a concrete universe we give a way to tie the knot:

```
data Close (f :: (* → *) → * → *) a
type (f :: (* → *) → * → *) :+: (g :: (* → *) → * → *) :: (* → *) → * → *
```

Using which we can construct a concrete universe:

```
type MyUniverse = Close (A0 Int :+: A1 Maybe :+: A2 Either)
```

Elements of the universe can be constructed using smart constructors:

```
int     :: A0 Int u :< u      ⇒ u Int
maybe :: A1 Maybe u :< u ⇒ u a → u (Maybe a)
(-->)  :: A2 (→) u :< u     ⇒ u a → u b → u (a → b)
...
```

In order to add a new smart constructor we need only use the injection from the $:<$ type class.

```
int = inject A0
maybe a = inject (A1 a)
a --> b = inject (A2 a b)
...
```

Finally, a library can be created which is polymorphic in the representation of types:

```
data Item = Item String (TypedValue u)
data Library u = Library String [Item u]
```

We can now put everything together to create an example of a very simple library, containing only addition on integers.

```
myLibrary :: (A0 Int u :< u, A2 (→) u :< u) ⇒ Library u
myLibrary = Library "My Library" [Item "+" ((+) ::: int --> int --> int)]
```

With the library in place we can move on to interpreting programs written in the DSL.

3 Type-safe interpretation

We use our *TypedValue*s to expose EDSLs to the outside world in a type-safe and useful way. The previous section showed how we can gather a group of functions into a *Library*. In this section we construct a type-safe function (*interpret*), which interprets a string as a program written in a small functional language and defined in terms of the *Library* functions. We start with the definition of a datatype for expressions, which we limit to the essentials for discussion purposes. The type *UExpr* (U for "untyped") contains constructors for variables (*UVar*), application (*UApp*), and lambda-abstraction (*ULam*).

```
data UExpr where
  UVar :: String → UExpr
  UApp :: UExpr → UExpr → UExpr
  ULam :: String → UExpr → UExpr
```

If we attempt to write an interpreter for expressions of this type we quickly run into trouble, as it's unclear how we should interpret the lambda case (*ULam*):

```
interpret :: (TypeEquality tr, A2 (→) tr :< tr)
          ⇒ Env tr → UExpr → Maybe (TypedValue tr)
interpret en e = case e of
  UVar v   → en v
  UApp f x → do
    f' ← interpret en f
    x' ← interpret en x
    app f' x'
  ULam v e →
    let a   = _  -- should be the type of v
        b   = _  -- should be the type of e
        fun x = fromJust (interpret (extend en v (x ::: a)) e)
    in return (fun ::: a --→ b)
type Env tr = String → Maybe (TypedValue tr)
```

The first case in the definition is self-explanatory. The second case uses the *app* function below to apply one *TypedValue* to another.

```
app :: forall u. (TypeEquality u, A2 (→) u :< u)
    ⇒ TypedValue u → TypedValue u → Maybe (TypedValue u)
app (f ::: funType) (x ::: arg) = do
  A2 from to :: A2 (→) u _f ← eject funType
  Refl ← from ?= arg
  return (f x ::: to)
```

The third case is where things get interesting. How do we interpret lambdas? Clearly, lambda expressions should be interpreted as functions from some type a to some type b, but how should we choose a and b? The problem is that we

do not have access to the type of the result or the argument of the lambda. There is no fundamental reason why this has to be the case, type inference for the simply typed lambda calculus with monomorphic constants is known to be a solved problem! From an untyped expression and a library we can easily derive a typed expression. Below is an encoding of the typed expressions:

```
data Expr tr where
  Var :: String → Expr tr
  App :: Expr tr → Expr tr → Expr tr
  Lam :: String → tr a → Expr tr → tr b → Expr tr
```

A value *Lam x ra e rb* represents a lambda expression $(\lambda x \to e) :: a \to b$ where $ra :: tr\ a$ and $rb :: tr\ b$. Using a standard type inference algorithm, like algorithm W [5], it is possible to infer the type annotations in an *Expr tr* from just an untyped *UExpr* and a *Library tr*. With type annotations in place, we can fix our interpreter to act correctly in the case for *Lam*.

```
interpret ::    (TypeEquality u, A2 (→) u :< u)
          ⇒     Env u → Expr u → Maybe (TypedValue u)
interpret en e = case e of
  Var v       → en v
  App f x     → do
    f' ← interpret en f
    x' ← interpret en x
    app f' x'
  Lam v a e b →
    let fun x  = let en' = extend en v (x ::: a)
                     Just res = interpret en' e
                 in  fromJust (coerce res b)
    in return (fun ::: a --→ b)
```

Note the use of *fromJust* in the last row of the definition of *fun*. This is an ostensibly partial operation, but we claimed to have provided a *type-safe* interpreter! Not to worry, the interpreter always returns a *Just v* given a correct environment and a well-typed expression *e*. This means that we can safely use *interpret* in a context where expressions are type-checked before we call it. This means that we can build a safe function $run :: Complete\ tr \Rightarrow Library\ tr \to String \to Maybe\ (TypedValue\ tr)$, where we have:

```
type Complete tr = (TypeEquality tr, A2 (→) tr :< tr)
```

We elide the functions for parsing and type checking but present *run*:

```
parse     :: String → Maybe UExpr
typeCheck :: Complete tr ⇒ Library tr → UExpr → Maybe (Expr tr)
libToEnv  :: Library tr → Env tr

run       :: Complete tr ⇒ Library tr → String → Maybe (TypedValue tr)
run l s = parse s >>= typeCheck l >>= interpret (libToEnv l)
```

It would be possible to take one further step towards manifest type-safety: we could add type-indices to *Expr* and *Env* to obtain an interpreter without *Maybe*. If we ignore variable binding this would result in an interpreter in the style of $eval1 :: E\ a \to a$ for a type indexed expression type E. With binding, we would also need a parameter for the environment: $eval2 :: env \to E\ env\ a \to a$. But in our full setting, with parameterisation also over the universe, we decided this would take us too far from the intended application and leave it as future work.

4 Case study: GRACe

The GRACe language [11] is a Haskell EDSL for working with diagrammatic systems of components implemented by constraint logic programming. The GRACe DSL is used in the GRACeFUL RAT [15] (Rapid Assessment Tool), which is a tool for graphical composition of maps representing causal relationships between parts of complex systems, such as systems describing urban design and its sensitivity to weather. The rapid assessment tool consists of a Haskell backend connected to a constraint solver and a web-based visual editor frontend [13]. The tool exposes a library of functions written in the DSL to the user as graphical widgets.

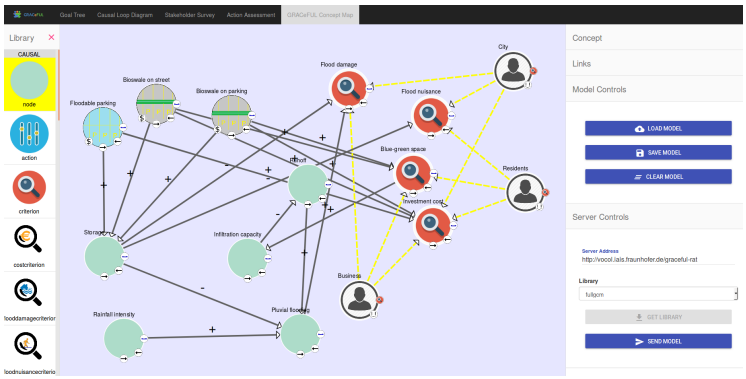


Fig. 3. A GRACe graph representation in the visual editor frontend.

The function widgets have parameter fields for the user to specify the function parameters, as well as input and output ports that can be connected to send the output of one function as the input to another. Using these widgets, the user constructs a program in the form of a graph. The graph is submitted to the Haskell backend using HTTP, at which point it is interpreted as a program in the GRACe EDSL.

The tool makes use of Typed Values to retain type information for the library functions exposed to the frontend which allows type-safe interpretation of the programs sent back from the frontend. The library contains a Typed Value

representing each of the exposed functions, along with metadata concerning the visual presentation of the function on the frontend. When the program graph is submitted to the backend the appropriate library functions are applied to parameters provided by the user, which are communicated in a type-safe manner using Typed Values. The resulting value is sent back to the frontend to be displayed to the user.

Using Typed Values we can prevent users from causing type errors by making mistakes like giving function parameters of the wrong type or sending the output of one function as the input to a function with a different input type. We use the Typed Value tags to annotate the library of functions with metadata to specify how the functions should be presented on the frontend.

5 Related work

In [17], Peyton Jones et al. present *Dynamic* and *TypeRep*, implemented with built-in support in GHC. We use a similar technique but can unfortunately not use their *TypeRep* directly because we want support for "labelling" of type representations to help communication with the external world. They support an open universe of all Haskell types rather than our "extensible but closed" universes.

In his lecture notes on "Typed Tagless Final Interpreters" [12], Kiselyov presents a technique (or design pattern) for representing typed higher-order languages (DSLs) in a typed metalanguage (Haskell), along with type-preserving interpretation. It is a powerful technique, but does not deal with connecting to the untyped world as we do.

Baars and Swierstra [2] present an approach for dynamic typing that is similar to the standard libraries *Dynamic* and *Typeable*, but they abstract over the actual type representation using an type class. Whereas they abstract over the type representation, we show how to control the range of type representation using an universe of types. We can extend an existing universe of types, instead of supplying a new instance for a type representation type class. The paper by Baars and Swierstra [2] also shows how to construct a typed evaluator that can interpret expressions using dynamic typing. The main difference is that the supported expressions need to be tagged with their types. We don't need to have such annotations since we infer the type of an expression.

Cheney and Hinze [4] use the same type representation encoding, but focus their library more on generic programming, rather than dynamic type checking. Similarly Bahr and Hvitved [3] present a compositional encoding of datatypes with an emphasis on recursion schemes. They do not focus their effort on encoding type representations, however.

6 Conclusions and future work

We have presented a framework for exposing Haskell EDSLs to the untyped world and interpreting the resulting EDSL programs in a type-safe manner.

We have shown how this framework is useful in a small example (Henderson's functional geometry EDSL) as well as a larger real-world case study (GRACe). The mechanisms for achieving this have all been implemented in Haskell without special compiler support.

Future work Currently our technique only supports exposing monomorphic APIs; supporting polymorphic APIs is noted as future work. Achieving this would be both useful and technically interesting. To the best of our understanding, representing polymorphic types without special compiler support is a non-trivial task. Being able to expose polymorphic EDSLs using our technique would significantly increase the versatility of the language.

It would also be interesting to explore the extent to which our technique for adding annotations could be used to add semantically rich annotations. The tagging mechanism that allows us to attach additional documentation to a type in our *TypedValues* could be extended to express contracts in the style of Hinze et al. [10], stating properties that the attached value must satisfy. Adding contracts could potentially greatly increase the utility of the framework for the EDSL writer. Being able to specify pre- and post-conditions is useful both for the EDSL writer and the end user.

It would be interesting to evaluate the Saint library both in terms of usability (can new users easily apply it to their EDSLs) and efficiency (how much overhead, in memory and time, is used by Saint).

Finally it would be a natural direction to continue up the ladder of type safety to a type indexed expression datatype and a tag-less interpreter.

Acknowledgements

This work was partially supported by the projects GRACeFUL (grant #640954) and CoeGSS (grant #676547), which have received funding from the European Union's Horizon 2020 research and innovation programme. It was also partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

References

1. Axelsson, E., et al.: Feldspar: A domain specific language for digital signal processing algorithms. In: 8th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010). pp. 169–178. IEEE (2010). https://doi.org/10.1109/MEMCOD.2010.5558637
2. Baars, A.I., Swierstra, S.D.: Typing dynamic typing. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming. pp. 157–166. ICFP '02, ACM, New York, NY, USA (2002). https://doi.org/10.1145/581478.581494
3. Bahr, P., Hvitved, T.: Compositional data types. In: Proceedings of the seventh ACM SIGPLAN workshop on Generic programming. pp. 83–94. WGP '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2036918.2036930

4. Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. pp. 90–104. Haskell '02, ACM, New York, NY, USA (2002). https://doi.org/10.1145/581690.581698
5. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages. pp. 207–212. POPL '82, ACM, New York, NY, USA (1982). https://doi.org/10.1145/582153.582176
6. Eisenberg, R.A., Weirich, S.: Dependently typed programming with singletons. In: Proceedings of the 2012 Haskell Symposium. pp. 117–130. Haskell '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2364506.2364522
7. Heeren, B., Jeuring, J., Gerdes, A.: Specifying rewrite strategies for interactive exercises. Mathematics in Computer Science **3**(3), 349–370 (2010). https://doi.org/10.1007/s11786-010-0027-4
8. Henderson, P.: Functional geometry. In: Proceedings of the 1982 ACM Symposium on LISP and Functional Programming. pp. 179–187. LFP '82, ACM, New York, NY, USA (1982). https://doi.org/10.1145/800068.802148
9. Henderson, P.: Functional geometry. Higher-Order and Symbolic Computation **15**(4), 349–365 (Dec 2002). https://doi.org/10.1023/A:1022986521797
10. Hinze, R., Jeuring, J., Löh, A.: Typed contracts for functional programming. In: Hagiya, M., Wadler, P. (eds.) Functional and Logic Programming, 8th International Symposium, FLOPS 2006. LNCS, vol. 3945, pp. 208–225. Springer (2006). https://doi.org/10.1007/11737414_15
11. Jansson, P., et al.: D4.2: A domain specific language for GRACeFUL concept maps (2017), `https://github.com/GRACeFUL-project/DSL-WP/raw/master/deliverables/d4.2.pdf`, deliverable of the GRACeFUL project (640954)
12. Kiselyov, O.: Typed tagless final interpreters. In: Gibbons, J. (ed.) Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming. pp. 130–174. SSGIP'10, Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32202-0_3
13. Krishna Murthy, D.R., Wiens, V., Lohmann, S., Asmat, R.: D3.3: VA EDA tool prototype (2017), deliverable of the GRACeFUL project. FETPROACT-1-2014 Grant No 640954
14. Lämmel, R., Peyton Jones, S.: Scrap your boilerplate: A practical design pattern for generic programming. In: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. pp. 26–37. TLDI '03, ACM, New York, NY, USA (2003). https://doi.org/10.1145/604174.604179
15. Lohmann, S.: D2.5: CRUD RAT prototype (2017), deliverable of the GRACeFUL project. FETPROACT-1-2014 Grant No 640954
16. Mestanogullari, A., Hahn, S., Arni, J.K., Löh, A.: Type-level web APIs with Servant: An exercise in domain-specific generic programming. In: Proc. 11th ACM SIGPLAN Workshop on Generic Programming. pp. 1–12. ACM (2015). https://doi.org/10.1145/2808098.2808099
17. Peyton Jones, S., Weirich, S., Eisenberg, R.A., Vytiniotis, D.: A reflection on types. In: Lindley, S., et al. (eds.) A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, pp. 292–317. Springer (2016). https://doi.org/10.1007/978-3-319-30936-1_16
18. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (Jul 2008). https://doi.org/10.1017/S0956796808006758
19. Wadler, P.: The expression problem (1998), appeared on the Java-genericity mailing list. Available here: `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.