Type Safe Interpreters for Free

Maximilian Algehed Sólrún Halla Einarsdóttir Alex Gerdes Patrik Jansson

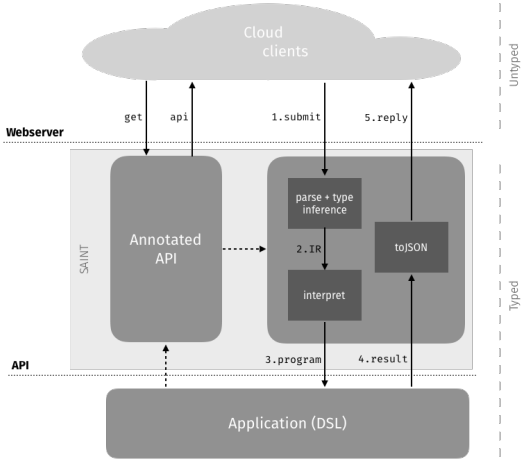
Functional Programming division, Chalmers University of Technology

2018-06-12 (TFP 2018)

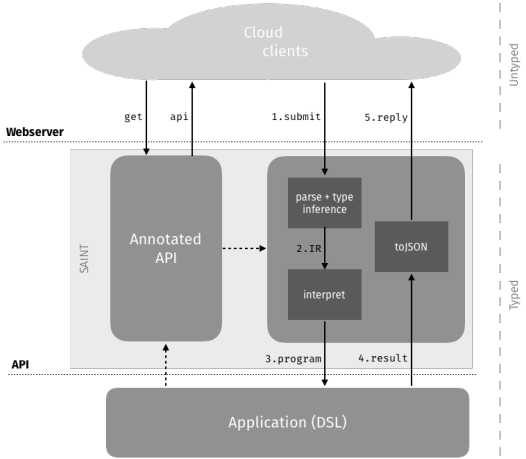
`https://github.com/GRACeFUL-project/Saint`

Acknowledgments: Funding from Horizon 2020 through GRACeFUL (grant #640954) and CoeGSS (grant #676547) and from Knut and Alice Wallenberg Foundation through WASP.

How “Saint” Connects Your EDSL with the Cloud



How “Saint” Connects Your EDSL with the Cloud



Henderson's Fish

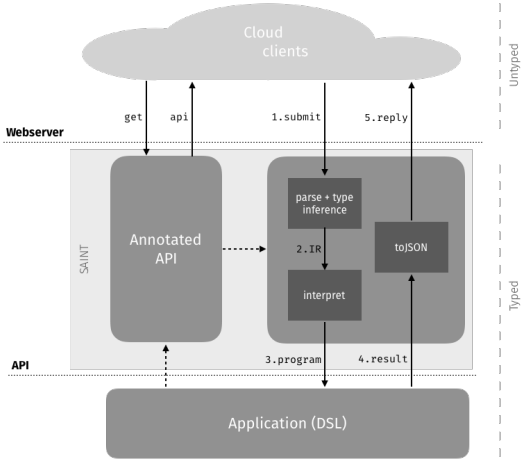
Text

```
scale 200 (rot fish)
```

Submit

Result

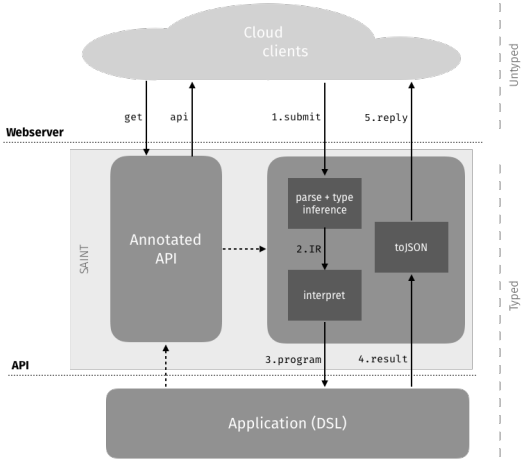
How “Saint” Connects Your EDSL with the Cloud

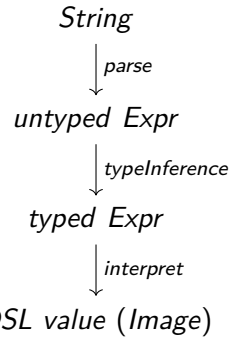


Text:

```
let sqrl = ...
    -- 10 more lines
in scale 100 (sqrl 3)
```

How “Saint” Connects Your EDSL with the Cloud





How “Saint” Connects Your EDSL with the Cloud

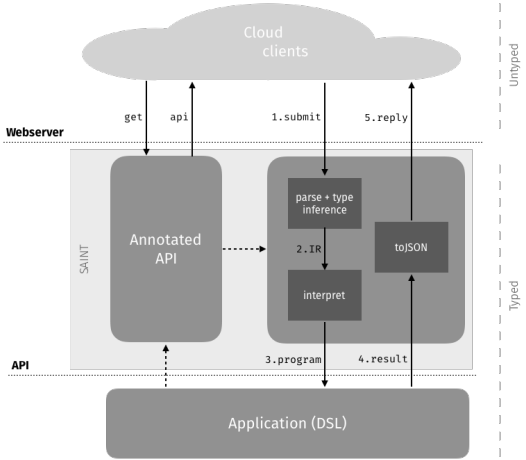
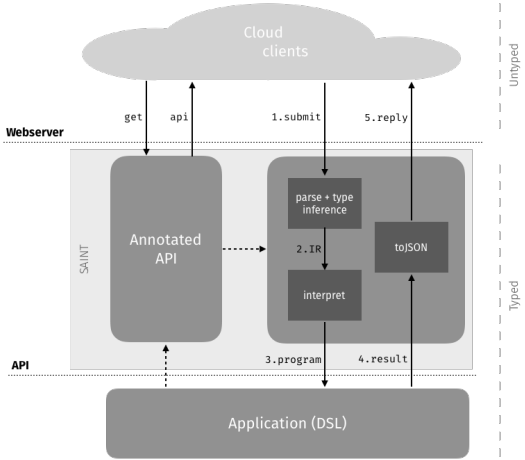
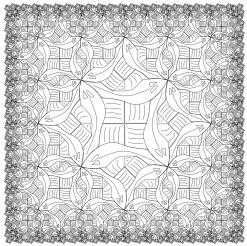


Image: first 14000 splines, then raw PNG data

How “Saint” Connects Your EDSL with the Cloud





Example DSL: Henderson's functional geometry (fish)

```
beside :: Image → Image → Image
above  :: Image → Image → Image
over   :: Image → Image → Image                  -- overlay
rot    :: Image → Image                          -- 90 degrees
natrec :: Image →                                -- base case
          (Int → Image → Image) →                -- step function
          Int →
          Image
fish   :: Image

data Image   -- just a list of splines
```

Example: the fish DSL API as a *Library*

For Saint to help, we need to describe the API as a value.

```
fishLib :: Library
fishLib = Library "fish"
  [ Item "beside" $ beside ::: image --> image --> image
  , Item "above"  $ above  ::: image --> image --> image
  , Item "over"   $ over   ::: image --> image --> image
  , Item "rot"    $ rot    ::: image --> image
  , Item "natrec" $
      natrec :::
        image -->
        (int --> image --> image) -->
        int -->
        image
  , Item "fish"   $ fish ::: image
  ]
```

Example: the fish DSL API as a *Library*

For Saint to help, we need to describe the API as a value.

```
fishLib :: Library
fishLib = Library "fish"
  [ Item "beside" $ beside ::: image --> image --> image
  , Item "above"  $ above  ::: image --> image --> image
  , Item "over"   $ over   ::: image --> image --> image
  , Item "rot"    $ rot    ::: image --> image
  , Item "natrec" $
      natrec ::: Tag "Recursion over Nat"
        image -->
        Tag "The step function" (int --> image --> image) -->
        int -->
        image
  , Item "fish"   $ fish ::: Tag "The fish base image" image
  ]
```

Exposing the Library API

Any *lib* :: *Library* is a type-annotated lookup table.

```
data Library = Library String [Item]
data Item    = Item String TypedValue

data TypedValue where    -- basically Dynamic
  (:::) :: a → TRep a → TypedValue
infixr 0 :::
```

Codes for types (more general in the paper):

```
data TRep t where
  TImage :: TRep Image
  TInt   :: TRep Int
  TFun   :: TRep a → TRep b → TRep (a → b)
  Tag    :: String → TRep a → TRep a

image = TImage; int = TInt; (-->) = TFun
infixr 1 -->
```

Simple type representations

```
data TRep t where    -- Codes for types:
  TImage :: TRep Image
  TInt   :: TRep Int
  TFun   :: TRep a → TRep b → TRep (a → b)
  Tag    :: String → TRep a → TRep a
```

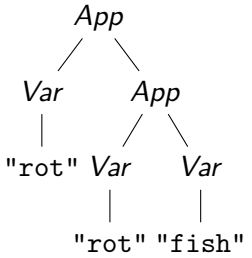
Working with (dynamic) typed values:

```
data a ≡ b where
  Refl :: a ≡ a

(≟) :: TRep a → TRep b → Maybe (a ≡ b)
  -- ≟ def. by simple syntactic equality (elided)

coerce :: TRep a → TypedValue → a
coerce a (x ::: b) = case a ≟ b of
  Just Refl → x
```

Example use of the interpreter

data *Expr* **where**
Var :: *String* → *Expr*
App :: *Expr* → *Expr* → *Expr*
Lam :: *String* → *TRep a* → *Expr* → *TRep b* → *Expr*

Example:



↦ *Just* (*rot* (*rot fish*) ::: *image*)

Towards a Type Safe Interpreter

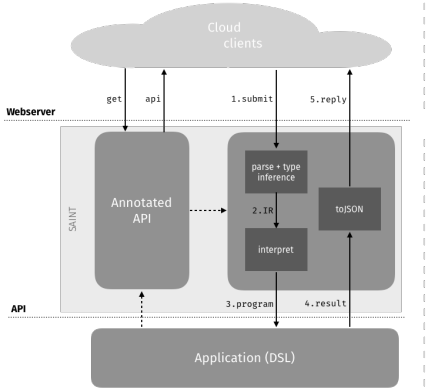
```
app :: TypedValue → TypedValue → Maybe TypedValue
app (f ::: TFun a b) (x ::: a') = do
  Refl ← a ≟ a'
  return (f x ::: b)
app _ _ = Nothing

data Expr where
  Var  :: String → Expr
  App :: Expr → Expr → Expr
  Lam :: String → TRep a → Expr → TRep b → Expr

type Env = String → Maybe TypedValue          -- or similar
extend   :: String → TypedValue → Env → Env   -- simple
libToEnv :: Library → Env                     -- also simple
```

Type Safe Interpreter

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

interpret :: Env → Expr → Maybe TypedValue
interpret env e = case e of
  Var v     → env v
  App f a → do    f' ← interpret env f
                  a' ← interpret env a
                  app f' a'
  Lam v t bo t' → return (lam ::: (t --→ t'))
    where lam x = let env' = extend v (x ::: t) env
                      Just res = interpret env' bo
                  in  coerce t' res
```

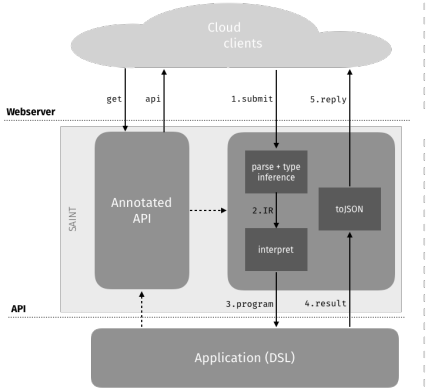
Saint paper summary

- a framework (Saint) for exposing a typed API to an untyped world



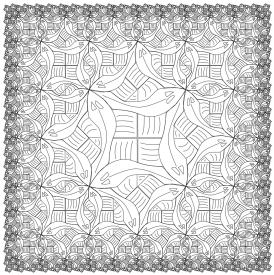
Saint paper summary

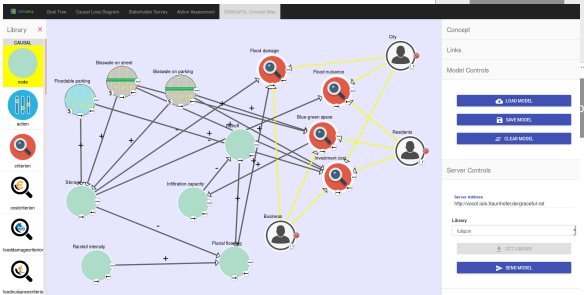
- a framework (Saint) for exposing a typed API to an untyped world
- a version of *Typeable* supporting tags (annotations in the *TRep*)
- a generic, type safe interpreter in Haskell

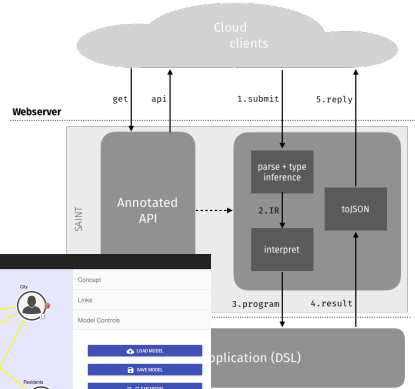


Saint paper summary

- a framework (Saint) for exposing a typed API to an untyped world
- a version of *Typeable* supporting tags (annotations in the *TRep*)
- a generic, type safe interpreter in Haskell
- two case studies: FISH and GRACe







Questions?

Type Safe Interpreters for Free

Maximilian Algehed Sólrún Halla Einarsdóttir Alex Gerdes Patrik Jansson

Functional Programming division, Chalmers University of Technology

2018-06-12 (TFP 2018)

https://github.com/GRACeFUL-project/Saint

Acknowledgments: Funding from Horizon 2020 through GRACeFUL (grant ∉640954) and CoreGSS (grant ∉676547) and from Knut and Alice Wallenberg Foundation through WASP.

How "Saint" Connects Your EDSL with the Cloud