Higher-Order Intensional Type Analysis

Stephanie Weirich
Cornell University

Reflection

- A style of programming that supports the *run-time discovery* of program information.
  - “What does this code do?”
  - “How is this data structured?”
- Running program provides information about itself.
  - self-descriptive computation.
  - self-descriptive data.

Applications of reflection

- **Runtime systems:** garbage collection, serialization, structural equality, cloning, hashing, checkpointing, dynamic loading
- **Code monitoring tools:** debuggers, profilers
- **Component frameworks:** software composition tools, code browsers
- **Adaptation:** stub generators, proxies
- **Algorithms:** iterators, visitor patterns, pattern matching, unification

What is reflection?

- **Run-time examination of type or class.**
- Not dynamic dispatch in OO languages.
  - Have to declare an instance for every new class declared. Easy but tedious.
  - Simple apps hard-wired in Java.
- Not instanceof operator in OO languages.
  - It requires a closed world.
    - Need to know the name of the class a priori.
    - Need to know what that name means.

Structural Reflection

- Need to know about the structure of the data to implement these operations once and for all.
- Intensional Type Analysis
  - Examines the structure of types at run time.
  - A term called tcase implements case analysis of types.

Serialization

```
serialize[α] (x:α) =
  tcase α of
      int      ) int2string(x)
      string)  “\“” + x + “\””
      β ′ γ    ) “(” + serialize[β](x.1) + “,”
                          + serialize[γ](x.2)  + “)”
      β → γ )  “<function>”
```

State of the art

- No system for defining type-indexed functionality extends to both type constructors and quantified types.

Type constructors

- Types indexed by other types.
- Useful to describe parameterized data structures.
  - **head :8α. list α → α**
  - **tail :8α. list α → list α**
  - **add :8α. (α ′ list α) → list α**
- Don't have to cast the type of elements removed from data structures.

Type functions

- Type constructors are functions from types to types.
- Expressed like lambda-calculus functions.

$$\tau ::= \ \ldots \mid \lambda\alpha\,.\tau \ \mid \tau_1\,\tau_2 \mid \alpha$$

- *Example:*

$$\mathbf{Quad} = \lambda\alpha.\ (\,\alpha\,'\,\alpha\,)\,'\,(\,\alpha\,'\,\alpha\,)$$

- Static language for reasoning about the relationship between types.

Types with binding structure

- Parametric polymorphism hides the types of inputs to functions.

  $$8\boldsymbol{\alpha}.\ \boldsymbol{\alpha} \rightarrow \textbf{string}$$

- *Other examples*:
  - Existential types ($\exists\alpha\ .\ \tau$) hide the actual type of stored data.
  - Recursive types ($\mu\alpha.\ \tau$) describe data structures that may refer to themselves (such as lists).
  - Self quantifiers ($\textbf{self}\ \alpha.\ \tau$) encode objects.

Problems with these types

- tcase is based on the fact that the closed, simple types are *inductive*.

  $\tau ::= \mathbf{int}\ |\ \mathbf{string}\ |\ \tau1 \rightarrow \tau2\ |\ \tau1\ '\ \tau2$

- Analysis is an iteration over the type structure.
- With quantified types, the structure is not so simple.

  $\tau ::= \ldots |\ 8\alpha.\ \tau\ \ |\ \alpha$

Example

```
tcase α  of
  int       ) …
  string ) …
  β → γ )    …
  β ′ γ    ) …
  8α.??  ) …
```

Here β and γ are bound to the subcomponents of the type, so they may be analyzed.

Can’t abstract the body of the type here, because of free occurrences of α.

Higher-order abstract syntax

- Type constructors for polymorphic types.

$$8\alpha\,.\,\alpha \rightarrow \alpha \quad \textbf{vs.}\ 8(\lambda\,\alpha\,.\,\alpha \rightarrow \alpha\,)$$

- 8 branch abstracts that constructor.

```
typecase  8(λ α . α → α ) of
    int      )  e1
    β → γ  )  e2
    8δ       )  e3
reduces to  e3 with δ  replaced by (λ α . α → α )
```

- Have to apply **δ** to some type in order to analyze it.

[Trifonov et al.]

Works for some applications

```
serialize[α] (x:α) =
    tcase α of
        int      ) int2string(x)
        string  )  “\“” + x + “\””
        β ′ γ    ) “(” + serialize[β](x.1) + “,”
                          + serialize[γ](x.2)  + “)”
        β → γ  )  “<function>”
        8δ       ) “<polymorphic function>”
        ∃δ       ) let <β, y> = unpack x in
                           serialize [δ(β)] y
```

But not for all

```
serializeType[α] =
    tcase α of
        int      ) “int”
        β ′ γ    )  “(”   + serializeType[β] + “ * ”
                          + serializeType[γ] + “)”
        β → γ ) “(”      + serializeType[β] + “ -> ”
                          + serializeType[γ] + “)”
        8δ       ) ???
        ∃β       ) ???
```

Two solutions with one stone

If we can analyze
type constructors
in a principled way,

then we can analyze
quantified types
in a principled way.

Type equivalence

- For type checking, we must be able to determine when two types are semantically equal.
  - to call a function the argument must have an equivalent type.
- *Reference algorithm*: fully apply all type functions inside the two types and compare the results.

$$(\lambda\ \alpha.\ \alpha\ {}^{\prime}\ \alpha)\ (\mathrm{int}) =?\ (\lambda\ \beta.\ \beta\ {}^{\prime}\mathrm{int})\ (\mathrm{int})$$

$$\mathrm{int}^{\prime}\ \mathrm{int} =?\ \mathrm{int}\ {}^{\prime}\ \mathrm{int}$$

Constraint on type analysis

- When we analyze this type language we *must* respect type equivalence.

tcase [(λα. α ′ int) int]…

must produce the same result as

tcase [int ′ int]…

- Type functions, applications, and variables must be “transparant” to analysis.

Generic/Polytypic programming

- Generates operations over parameterized data-structures. [Moggi&Jay][Jansson&Juering][Hinze]
  - Example: **gmap<list>** applies a function f to all of the **α**'s in **list α.**
- *Compile-time* specialization. No type information is analyzed at run-time.
  - Can't handle polymorphic or existential types.

Idea

- A polytypic definition must also respect type equality.
  - $\mathbf{foo} < (\lambda\alpha.\ \alpha\ '\ \mathbf{int})\ \mathbf{int} > \ = \mathbf{foo} < \mathbf{int}\ '\ \mathbf{int} >$

- Produce equivalent terms for equivalent types.
  - $\mathbf{foo} < ((\lambda\alpha.\ \alpha\ '\ \mathbf{int})\ \mathbf{int} > = (\lambda\ x.\ x + 1)\ 1$
  - $\mathbf{foo} < \mathbf{int}\ '\ \mathbf{int} > = 1 + 1$

Idea

- Create an *interpretation* of the type language with the term language.
  - Map type functions to term functions.
  - Map type variables to term variables.
  - Map type applications to term applications.
  - Map type constants to (almost) anything.
- We can use this idea at run-time to analyze type constructors and quantified types.

Type Language

t ::= α
| λα. τ
| $\tau_1\ \tau_2$
| int | string
| → | ′ | 8

- variable
- function
- application
- constants

- **The type int ′ int is the constant ′ applied to int twice.**
- **The type 8α . α →α is the constant 8 applied to the type constructor (λα . α →α).**

Interpreter

Instead of tcase, define analysis term:

$$\text{tinterp}[\eta]\ \tau$$

- To interpret this language we need an environment to keep track of the variables.
- This environment will also have mappings for all of the constants.

Operational semantics of tinterp

- Type constants are retrieved from the environment

tinterp[η] int → η(int)
tinterp[η] string → η(string)
tinterp[η] → → η(→)
tinterp[η] ′ → η(′)
tinterp[η] 8 → η(8)

- Type variables are retrieved from the environment

tinterp[η] α → η(α)

Type functions

- Type functions are mapped to term functions.
- When we reach a type function, we add a new mapping to the environment.

$$\textbf{tinterp}[\eta]\ (\lambda\alpha.\tau)\ \rightarrow$$

$$\lambda\, x.\ \textbf{tinterp}[\underbrace{\eta+\{\alpha)x\}}_{\text{Execution extends environment, mapping } \alpha \text{ to } x.}]\ (\ \tau\ )$$

Application

- Type application is interpreted as term application

$\text{tinterp}[\eta]\ (\tau_1\ \tau_2)$

$\Rightarrow (\text{tinterp}[\eta]\ \tau_1)\ (\text{tinterp}[\eta]\ \tau_2)$

The interpretation of τ_1 is a function

Example

```
serializeType = tinterp [η]
where η =  {
    int        ) “int”
    string   ) “string”
    ′            )  λ x:string. λ y:string.
                      “(” + x + “*” + y + “)”
    →          )  λ x:string. λ y:string.
                      “(” + x + “->” + y + “)”
    8           )  λ x:string→string.
                      let v = gensym () in
                      “(all ” + v + “.” + (x v) + “)”
}
```

Example execution

serializeType[int′int]

➔ (tinterp[η] ′) (tinterp[η] int) (tinterp[η] int)

➔ (λ x:string. λ y:string. "("+ x +"*"+ y +")")
 (tinterp[η] int) (tinterp[η] int)

➔ (λ x:string. λ y:string. "("+ x +"*"+ y +")")
 "int" "int"

➔ "(" + "int" + "*" + "int" + ")"

➔ "(int*int)"

Example

```
serializeType = tinterp [η]
where η =  {
    int        ) "int"
    string   ) "string"
    ′            )  λ x:string. λ y:string.
                       "(" + x + "*" + y + ")"
    →          )  λ x:string. λ y:string.
                       "(" + x + "->" + y + ")"
    8            )  λ x:string→string.
                       let v = gensym () in
                       "(all " + v + "." + (x v) + ")"
}
```

Not the whole story

- More complicated examples require a generalization of this framework.
  - Must allow the type of each mapping in the environment to depend on the analyzed type.
  - Requires maintenance of additional type substitutions to do so in a type-safe way.
  - This language is type sound.
- Details appear in paper.

Conclusion

- Reflection is analyzing the structure of abstract types.
- Branching on type structure doesn’t scale well to sophisticated and expressive type systems.
- A better solution is to interpret the compile-time language at run-time.