

Programming with Types

Run-time type
analysis and the
foundations of
program reflection

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

Primitive notions of reflection

- What is the fundamental enabling mechanism to support 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.
- Java Reflection API
 - Classes to describe the type structure of Java Class, Field, Method, Array,...
 - Methods to provide access to these classes at run time: Object.getClass, Class.getFields, Field.getType ...

```
String serialize( Object o ) {
    String result = "[";
    Fields[] f = o.getClass().getFields();
    for ( int i=0; i<f.length; i++ ) {
        Class fc = f[i].getType();
        if ( fc.isPrimitive() ) {
            if ( fc == Integer.TYPE ) {
                result += serializeInt((Integer) f[i].get( o ) );
            } else if ( fc == Boolean.TYPE ) {
                result += serializeBoolean((Boolean) f[i].get( o ) );
            } else if ...
        } else { result += serialize( f[i].get( o ) ); }
    }
    return result + "]" ;
}
```

Not integrated with type system

- Can't catch bugs statically.

```
if ( fc == Integer.TYPE )  
    result += serializeBoolean( (Boolean) f[i].get( o ) );
```

- Need redundant tests of type information.

```
if ( fc == Integer.TYPE )  
    result += serializeInt( (Integer) f[i].get( o ) );
```

- All objects must have attached type information.

```
o.getClass();  
(Integer)o;
```

Separating types from data

- Implementation must store type information with each data value.
 - Necessary for `getClass` and `runtime casts`.
- Can't express the run-time behavior of type information.
 - Hinders optimization in typed low-level languages.
- Prevents type abstraction in high-level languages.
 - Impossible to hide the implementation of an abstract data-type.
 - Necessary for modularity and representation independence.

Foundational study of reflection

- It is not clear how to smoothly integrate these dynamic mechanisms into a statically typed language.
- An ideal framework...
 - Must be connected with the type system.
 - Must be able to express optimizations.
 - Must allow type abstraction.
 - Must extend to advanced type systems.

My Work

- Examination of the foundational mechanisms for reflection.
 - Done in the context of typed lambda calculi
- Contributions in this area:
 - An accurate connection between run-time type information and types [Crary, Weirich, Morrisett 98].
 - A core reflection language with the flexibility to describe a variety of type systems [Crary & Weirich 99].
 - An encoding of these languages into a language without specialized reflection mechanisms [Weirich 01].
 - An extension of reflection that encompasses type constructors and quantified types [Weirich 02].

Formalizing reflection

- A standard typed lambda calculus plus an abstract datatype (ADT) to represent type information.

$\tau ::= \text{int} \mid \text{string} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \text{ ' } \tau_2 \mid \text{any} \mid \text{rep}$
 $e ::= 0 \mid 1 \mid \text{"foo"} \mid \dots \mid x \mid \lambda x:\tau. e \mid e_1 (e_2) \mid \langle e_1, e_2 \rangle \mid e.1 \mid e.2$

$(\tau) e \mid \text{typeof}(e)$
 $\text{Rint} \mid \text{Rstring}$
 $\text{Rarrow}(e_1, e_2) \mid \text{Rpair}(e_1, e_2)$
 $\text{tcase } e \text{ of}$
 $\text{Rint}) \dots$
 $\text{Rstring}) \dots$
 $\text{Rarrow}(x, y)) \dots$
 $\text{Rpair}(x, y)) \dots$

The type of
Some way to
hide the

A convenient way
to get the type of a
value

and a
checked cast
to recover it

Comparison with Java Reflection

Idealized Language

- any
- rep
- Rint
- Rstring
- typeof(e)
- $(\tau)e$

Java Reflection API

- Object
- Class/Field/Method
- Integer.TYPE
- String.getClass();
- e.getClass();
- *(classname)e*

Serialization

serialize has type: `any → string`

`serialize (x) =`

`tcase (typeof(x)) of`

`Rint) int2string((int) x)`

`Rstring) “\“” + (string) x + “\””`

`Rpair(w,z))`

`“(“ + serialize((any'any) x.1) + “,”`

`+ serialize((any'any) x.2) + “)”`

`Rarrow(w,z)) “<function>”`

Serialize without typeof

New type of serialize : rep ' any → string

serialize (xrep, x) =

tcase (xrep) of

Rint) int2string((int) x)

Rstring) “\“” + (string) x + “\””

Rpair(w,z))

“(“ + serialize(w, (any'any) x.1) + “,”

+ serialize(z, (any'any) x.2) + “)”

Rarrow(w,z)) “<function>”

Accurate reflection

- Connect types and their representations.
- A term has the type $\text{rep}(\tau)$ if it represents τ .

$\text{Rint} : \text{rep}(\text{int})$

$\text{Rstring} : \text{rep}(\text{string})$

$\text{Rpair}(e1, e2) : \text{rep}(\tau1 \text{ ' } \tau2)$ (if $e1 : \text{rep}(\tau1)$ and $e2 : \text{rep}(\tau2)$)

$\text{Rarrow}(e1, e2) : \text{rep}(\tau1 \rightarrow \tau2)$ (if $e1 : \text{rep}(\tau1)$ and $e2 : \text{rep}(\tau2)$)

- Type variables express the connection.
 - $x : \alpha, y : \text{rep}(\alpha)$

[Crary, Weirich, Morrisett 98]

Type Analysis

- The analysis term *refines* the type information.

tcase (x : rep(α)) of

Rint) ... α is int

Rstring) ... α is string

Rpair(e1, e2)) ... α is a pair type

Rarrow(e1,e2)) ... α is a function type

Benefits of this approach

- Can express low-level operation.
 - Rep types used to add dynamic loading to Typed Assembly Language (TAL).
[Hicks, Weirich, Crary 2000]
- Can optimize use of analysis.
 - **foo (x:array α , y:rep(α)) = tcase y of ...**
- Preserves type abstraction.
 - can't determine α without **rep(α)**

Scaling to more expressivity

- Current type systems are *much* more sophisticated.
 - Objects/Classes [Java, C++, C#, OCaml, ...]
 - First-class polymorphic/abstract types [Haskell, Cyclone, Vault, CLU, ...]
 - Higher-order type constructors [ML, Haskell, ...]
 - Region types [Cyclone, Vault, Tofte&Talpin, Gay&Aiken, ...]
 - Security types [JIF, MLIF, PCC, CCured, Cqual, Walker, ...]
 - Bounding time/space usage [Crary&Weirich]
 - Using resources correctly [Igarashi & Kobayashi, ...]
 - Dependent types [Cayenne, Xi, Shao et al., ...]
- Scaling structural type analysis to these systems in this framework is a challenge.

But we want to...

- These type systems are getting very good at describing the **behavior** of programs.
 - The goal of advanced type systems is to verify expressive program properties.
- Analyzing these types at run-time provides a foundation for *Behavioral Reflection*.
 - *Example*: if the type system tracks the running time of each method, a real-time scheduler may use this information.

Rest of Talk

- I will talk about how to extend type analysis to advanced type systems.
- Two crucial issues:
 - Type constructors
 - Types with binding structure
- These constructs are *foundational* to many current type systems.

Serialization

serialize $[\alpha]$ ($x:\alpha$) =

tcase α of

int) **int2string**(x)

string) “\” + x + “\””

β ' γ) “(” + **serialize** $[\beta]$ ($x.1$) + “,”
+ **serialize** $[\gamma]$ ($x.2$) + “)”

$\beta \rightarrow \gamma$) “<function>”

Type constructors

- Types indexed by other types.
- Useful to describe parameterized data structures.
 - **head** : $\delta\alpha. \text{list } \alpha \rightarrow \alpha$
 - **tail** : $\delta\alpha. \text{list } \alpha \rightarrow \text{list } \alpha$
 - **add** : $\delta\alpha. (\alpha ' \text{list } \alpha) \rightarrow \text{list } \alpha$
- Don't have to cast the type of elements removed from data structures.

Type functions

- Type constructors are functions from types to types.
- Expressed in the type syntax like lambda-calculus functions.

$$\tau ::= \dots \mid \lambda\alpha . \tau \mid \tau_1 \tau_2 \mid \alpha$$

- *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.

$\delta\alpha. \text{rep}(\alpha) ' \alpha \rightarrow \text{string}$

- *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 ($\text{self } \alpha. \tau$) encode objects.

Problems with these types

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

$$\tau ::= \text{int} \mid \text{string} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \mid \tau_2$$

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

$$\tau ::= \dots \mid \forall \alpha. \tau \mid \alpha$$

Example

```
tcase  $\alpha$  of  
  int ) ...  
  string ) ...
```

```
 $\beta \rightarrow \gamma$  ) ...
```

```
 $\beta' \gamma$  ) ...
```

```
 $\delta\alpha.??$  ) ...
```

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

- Use type constructors to represent polymorphic types.

$\delta \alpha . \alpha \rightarrow \alpha$ vs. $\delta(\lambda \alpha . \alpha \rightarrow \alpha)$

- In branch for δ , we can abstract that constructor.

tcase $\delta(\lambda \alpha . \alpha \rightarrow \alpha)$ of

int) ...

$\beta \rightarrow \gamma$) ...

$\delta\delta$) ... // δ is bound to $(\lambda \alpha . \alpha \rightarrow \alpha)$

- Have to apply δ to some type in order to analyze it. This works well for some examples.

[Pfenning&Elliot][Trifonov et al.]

But not for all

`serializeType[α] =`

`tcase α of`

`int) “int”`

`β ' γ) “(“ + serializeType[β] + “ * ”
+ serializeType[γ] + “)”`

`$\beta \rightarrow \gamma$) “(“ + serializeType[β] + “ -> ”
+ serializeType[γ] + “)”`

`8β) ???`

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 we must make sure that its argument has the right type.
- *Reference algorithm*: fully apply all type functions inside the two types and compare the results.

$$(\lambda \alpha. \alpha ' \alpha) (\text{int}) =? (\lambda \beta. \beta ' \text{int}) (\text{int})$$
$$\text{int}' \text{int} =? \text{int}' \text{int}$$

Constraint on type analysis

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

`tcase [$(\lambda\alpha. \alpha' \text{ int})$ int]...`

must produce the same result as

`tcase [int ' int]...`

- Type functions, applications, and variables must be “transparent” to analysis.
- Otherwise, execution of program depends on implementation of type checker.

Generic/Polytypic programming

- Provides a general way to generate 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 .
 - This is a *compile-time* specialization. No type information is analyzed at run-time.
- A polytypic definition must also respect type equality.
 - $\text{foo} < (\lambda\alpha. \alpha' \text{ int}) \text{ int} > = \text{foo} < \text{int}' \text{ int} >$

Basic 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

$t ::= \alpha$

| $\lambda\alpha. \tau$

| $\tau_1 \tau_2$

| **int** | **string**

| \rightarrow | ' | δ

- variable
- function
- application
- constants

- The type **int ' int** is the constant ' applied to **int** twice.
- The type **$\delta\alpha . \alpha \rightarrow\alpha$** is the constant δ applied to the type constructor **$(\lambda\alpha . \alpha \rightarrow\alpha)$** .

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

$\text{tinterp}[\eta] \text{ int} \quad \rightarrow \quad \eta(\text{int})$

$\text{tinterp}[\eta] \text{ string} \quad \rightarrow \quad \eta(\text{string})$

$\text{tinterp}[\eta] \rightarrow \quad \rightarrow \quad \eta(\rightarrow)$

$\text{tinterp}[\eta] ' \quad \rightarrow \quad \eta(')$

$\text{tinterp}[\eta] 8 \quad \rightarrow \quad \eta(8)$

- Type variables are retrieved from the environment

$\text{tinterp}[\eta] \alpha \quad \rightarrow \quad \eta(\alpha)$

Type functions

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

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

$\lambda x. \text{tinterp}[\underbrace{\eta + \{\alpha\}x}] (\tau)$

Execution extends
environment, mapping α to x .

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[τ] = tinterp [η] τ`

where $\eta = \{$

`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[τ] = tinterp [η] τ`

where $\eta = \{$

`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:

Stephanie Weirich. Higher-Order Intensional Type Analysis. In *European Symposium on Programming (ESOP '02)*.

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.

Future work

- Type-based reflection
 - Reconciliation of structural and name-based analysis.
- Multi-level programming
 - Extensible programming languages.
 - Domain-specific languages.
- Program verification
 - Sophisticated type systems allow the representation and verification of many program properties.

