Macro-based type providers in Scala

Eugene Burmako and Travis Brown

École Polytechnique Fédérale de Lausanne
University of Maryland, College Park

---

5 APRIL 2014

What are type providers, anyway?

What are type providers, anyway?

Compile-time metaprogramming facilities for "information rich programming"

What are type providers, anyway?

Compile-time metaprogramming facilities for "information rich programming"*

*to borrow a phrase from the F# community

A TYPE PROVIDER...

1. reads information from a data source
2. makes that information available to the program in types

Examples of information sources

- An XSD schema for XML
- A JSON-LD context
- A Web Service Description Language file
- SQL table definitions and stored procedures

Motivation

- You've got schemas that describe your data
- You want to use these descriptions in your code
- You *don't* want to repeat yourself!

Example: Schema bindings for RDF

```
@prefix dc: <http://purl.org/dc/terms/>.
@prefix dct: <http://purl.org/dc/dcmitype/>.
@prefix sga: <http://shelleygodwinarchive.org/>.
@prefix wiki: <https://en.wikipedia/wiki/>.

sga:ms-abinger-c57 a          dct:Text.
sga:ms-abinger-c57 dc:title   "Frankenstein Draft Notebook B"@en.
sga:ms-abinger-c57 dc:creator wiki:Mary_Shelley.
```

Example: Schema bindings for RDF

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix dc: <http://purl.org/dc/terms/>.

dc:title a rdf:Property;
  rdfs:comment "A name given to the resource."@en;
  rdfs:isDefinedBy dcterms:;
  rdfs:label "Title"@en;
  rdfs:range rdfs:Literal.

dc:creator a rdf:Property;
  # and on and on...
```

Example: Schema bindings for RDF

```
val frankensteinNotebookB = (
  URI("http://shelleygodwinarchive.org/ms-abinger-c57")
    .a(dct.Text)
    -- dc.title ->- "Frankenstein Draft Notebook B"
    -- dc.creator ->- URI(
      "https://en.wikipedia.org/wiki/Mary_Shelley"
    )
)
```

Example: Schema bindings for RDF

Follow along with example code and documentation:

<https://github.com/travisbrown/type-provider-examples>

We'll be using the W3C's Banana RDF library throughout:

<https://github.com/w3c/banana-rdf>

Low-tech solutions

Defining schema bindings manually

```
object dc extends PrefixBuilder("http://purl.org/dc/terms/") {
  val title = apply("title")
  val creator = apply("creator")
  // and on and on...
}
```

Defining schema bindings manually

```
object dc extends PrefixBuilder("http://purl.org/dc/terms/") {
  val title = apply("title")
  val creator = apply("creator")
  // and on and on...
}
```

But we're just repeating the RDF Schema we've seen above...

Defining schema bindings manually

- These vocabularies can be large (hundreds of terms)
- We're just repeating information from the RDF Schema
- We don't want to repeat ourselves!

Traditional solution: textual code generation

- Tied to a specific (often ad-hoc) build process
- Concatenating strings is unpleasant and error-prone
- Oblivious to semantics, e.g. dependencies between modules of the program
- Hard to customize
- Easy to get out of sync

Implementing type providers

- In F#: special support is built into the compiler
- In Scala: we can use the general purpose macro system

Implementing type providers

- In F#: special support is built into the compiler
- In Scala: we can use the general purpose macro system

...with Scala macros

- Anonymous type providers via def macros
- Public type providers via macro annotations

Anonymous type providers

In action

```
val dc = fromSchema("/dcterms.rdf")
```

In action

```
val dc = fromSchema("/dcterms.rdf")
```

That's all!

How it works

- Parses the schema resource
- Creates an instance of a structural type
- scalac figures out the rest

How it works

- Parses the schema resource at compile time
- Creates an instance of a structural type
- scalac figures out the rest

Generated code

```
val dc = new PrefixBuilder("http://purl.org/dc/terms/") {
  val title = apply("title")
  val creator = apply("creator")
  // et cetera...
}
```

Implemented with a macro

```
object PrefixGenerator {
  def fromSchema(path: String) = macro impl

  def impl(c: Context)(path: c.Expr[String]) = ...
}
```

Advantages of the anonymous approach

- Familiar syntax—just a method call
- Works in official Scala 2.10 and 2.11

Advantages of the anonymous approach

- Familiar syntax—just a method call
- Works in official Scala 2.10 and 2.11

Disadvantages

- Structural types don't work in Java
- Structural types involve reflective access in Scala

Advantages of the anonymous approach

- Familiar syntax—just a method call
- Works in official Scala 2.10 and 2.11

Disadvantages

- Structural types don't work in Java
- Structural types involve reflective access in Scala*

*but there's a partial workaround—see the example project

Public type providers

In action

```
@fromSchema("/dcterms.rdf") object dc extends PrefixBuilder
```

How it works

- Also parses the schema resource at compile-time
- Uses the provided object as a template
- Populates the object with generated members

Generated code

```
object dc extends PrefixBuilder("http://purl.org/dc/terms/") {
  val title = apply("title")
  val creator = apply("creator")
  // et ainsi de suite...
}
```

In comparison

```
// anonymous
val dc = new PrefixBuilder("http://purl.org/dc/terms/") {
  val title = apply("title")
  val creator = apply("creator")
}

// public
object dc extends PrefixBuilder("http://purl.org/dc/terms/") {
  val title = apply("title")
  val creator = apply("creator")
}
```

Also implemented with a macro

```
class fromSchema(path: String) extends StaticAnnotation {
  def macroTransform(annottees: Any*) = macro PrefixGenerator.impl
}

object PrefixGenerator {
  def impl(c: Context)(annottees: c.Expr[Any]*) = ...
}
```

Advantages of the public approach

- Generated code is straightforward and interoperable
- Provides a lot of notational freedom

Advantages of the public approach

- Generated code is straightforward and interoperable
- Provides a lot of notational freedom

Disadvantages

- Requires your users to depend on macro paradise
- Provides a lot of notational freedom

Summary

We can generate code from schemas

- Using def macros in vanilla Scala 2.10/2.11 (anonymous)
- Using macro annotations in macro paradise (public)

How practical is this? (Language support)

- Macro annotations aren't shipped in Scala 2.11
- No concrete plans to ship them in Scala 2.12
- This means anonymous type providers are more stable
- But they have important downsides, so it's a trade-off

How practical is this? (IDE support)

- Both anonymous and public type providers are whitebox
- This means limited supported in Intellij and Eclipse
- Also there's no easy way to look into macro expansions
- Or to generate scaladocs for generated code

How practical is this? (IDE support)

- Both anonymous and public type providers are whitebox
- This means limited supported in Intellij and Eclipse*
- Also there's no easy way to look into macro expansions*
- Or to generate scaladocs for generated code*

*this is something we are working on in Project Palladium

How practical is this? (Tool support)

- Build reproducibility is a solved problem
- Just don't go and talk to external data sources directly
- Use schemas that are fetched and versioned independently

Comparison with F#

- More raw power
- Limited IDE support
- Not yet part of the language standard

Summary

- Macros enable principled compile-time code generation
- Can successfully implement type providers
- Better support is necessary for optimal experience

Resources

- Our example project: <http://goo.gl/pSTZMx>
- Type providers in Scala: <http://goo.gl/s8cPlw>
- Project Palladium: <http://scalareflect.org/>

Or ask us!

- @xeno_by <xeno.by@gmail.com>
- @travisbrown <travisrobertbrown@gmail.com>

Thanks!