Let's Unify With Scala Pattern Matching! *

Edmund S.L. Lam[1] and Iliano Cervesato[1]

Carnegie Mellon University Qatar
sllam@qatar.cmu.edu and iliano@cmu.edu

Abstract

Scala's pattern matching framework supports algebraic data types. It also has an extensible system of user-definable pattern extractors. These advanced features enable the development of more complex term manipulations on top of Scala's primitive pattern matching infrastructure. In this paper, we discuss the development of a lightweight library for first-order term unification on top of this extensible pattern matching framework. Together with a set of combinators for writing unification control statements that resemble Scala pattern matching, this library serves as a basis for building more complex domain specific languages (e.g., constraint solvers, logical frameworks) that rely on unification.

1 Introduction

Scala [4] is a modern programming language with extensive support for both imperative and functional programming. Specifically, Scala's pattern matching framework supports algebraic data types through *case class definitions*. It also provides an extensible system of user-definable pattern extractors. These advanced features allow the programmer to build complex pattern matching routines on top of the language's native pattern matching framework (e.g., Joins and Actors [1]). Scala does not come with native support for unification, and the only open-source unification library for it [5] provides only basic utilities, which make little use of Scala's advanced features (e.g., flexible syntax, integration with pattern matching framework). In this paper, we build first-order term unification on top of this extensible pattern matching framework. We have developed a lightweight library in Scala that allows the programmer to perform unification over first-order terms. This library defines a set of combinators to implement control statements that resemble Scala's pattern matching. For instance, Figure 1 shows a unification example in the style of Scala's case matching statements.

Here, lines 1 and 2 declare `x` and `y`, two new logical variables, and line 3 defines `f` as the term `F(Const(5),x)`, where `F` is a subclass of `Term`, representing an uninterpreted binary function application. All are subclasses of the `Term` type. In line 4, term `f` calls its `unify` method with two input alternatives: lines 5–7 attempts to unify `f` and the term `Const(4)`, and if successful, the most general unifier θ is made available in the scope of the nested code sequence in line 6. Similarly, lines 8–10 defines another case that unifies `f` and

```
1  val x: Term = new LogVar()
2  val y: Term = new LogVar()
3  val f: Term = F(Const(5),x)
4  f unify (
5     Const(4) withMgu θ => {
6        ... //  there is no θ such that F(Const(5),x)θ = Const(4)θ
7     },
8     F(y,Const(4)) >=> {
9        ... //  executes with mgu [5/y, 4/x] implicitly applied to x and y
10    }
11 )
```

Figure 1: Code snippet highlighting unification case control statement

*This paper was made possible by grants NPRP 4-341-1-059 and JSREP 4-003-2-001, from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

```
1  val x: Term = new LogVar()
2  val y: Term = new LogVar()
3  val unifA = new Unif( Const(4) )
4  val unifB = new Unif( F(y,Const(4)) )
5  val f: Term = F(Const(5),x)
6  f match {
7     case unifA(θ) => ...
8     case unifB(θ) => ...
9  }
```

Figure 2: Code snippet highlighting unification in extensible pattern matching

`F(y,Const(4))`, but with an alternative programming flavor: if successful, `>=>` implicitly applies the most general unifier onto the logical variables `x` and `y` (logical variables double up as mutable containers), hence the instantiation of these logical variables are side-effects of the case statement. Like a matching case statement, these alternatives are tried in top-down order and the operation throws an exception if none applies. In the present scenario, the second case statement is executed. If, rather than using the case matching format, the programmer prefers to directly integrate his/her unification code with Scala's pattern matching framework, our unification library also includes a unification pattern extractor that allows this. The code fragment in Figure 1 then assumes the form shown in Figure 2.

Through Scala's extensible pattern matching framework, lines 3 and 4 declare two instances of pattern extractor `Unif` (whose implementation is discussed in Section 3) for the term `Const(4)` and `F(y,Const(4))`. These extractors are used in the matching statement (lines 7 and 8), corresponding respectively to unification cases in Figure 1. This lightweight library serves as a foundational basis for developing more complex domain specific languages (e.g., constraint solvers, logical frameworks) that rely on unification.

2 Abstract Semantics

In this section, we give an abstract semantics for our unification construct, implemented as the `unify` control statement. It describes the behavior of this construct independently of the underlying approach to implementation (a Scala embedding here). For simplicity, we focus on a core term language consisting of constants, variables and uninterpreted function symbols. We also omit typing information. We begin by introducing some notation. We write $\vec{o}$ for a tuple of syntactic objects o. We denote the extension of a sequence $\vec{o}$ with an object o as '$o, \vec{o}$', with () indicating the empty tuple. A generic substitution is denoted θ or ϕ. We write $o\theta$ for the application of θ on object o. The composition of two substitutions θ and ϕ is denoted by '$\theta \circ \phi$'. Given two first-order terms t_1 and t_2, the meta-operation $mgu(t_1, t_2)$ returns the most general unifier (mgu) θ of t_1 and t_2 if it exists, $\bot$ if not.

Standard expressions e_α Term expressions e_t Identifiers x Function symbols f

$$\begin{array}{rrcl} \text{Terms} & t & ::= & \texttt{Const}(e_\alpha) \mid \texttt{Var}(x) \mid f(\vec{e_t}) \\ \text{Substitutions} & \theta, \phi & ::= & \cdot \mid \theta, [t/x] \\ \text{Expressions} & e & ::= & e_\alpha \mid t \mid e_t \texttt{ apply } \theta \mid e_t \texttt{ unify } \{\vec{u}\} \\ \text{Unify Cases} & u & ::= & t \texttt{ withMgu } \theta \texttt{ => } e \mid t \texttt{ >=> } e \end{array}$$

Figure 3: Unification on Scala Terms: Abstract Syntax

Figure 3 shows the abstract syntax of the fragment of interest: we focus on our term language and the `unify` control statement. All other syntactic fragments of Scala (referred to as "standard expressions") are treated abstractly and denoted by e_α. Types are omitted for succinctness. Expressions that are expected to evaluate to terms are denoted by e_t.

Normal Form Expression n Standard Evaluations $\langle \Psi; e \rangle \mapsto \langle \Psi; e \rangle$

Unification Evaluations: $\langle \Psi; \theta; t \rangle \mapsto \langle \Psi; \theta; e \rangle$

$$\frac{\langle \Psi; e_\alpha \rangle \mapsto \langle \Psi'; e \rangle}{\langle \Psi; \theta; e_\alpha \rangle \mapsto \langle \Psi'; \theta; e \rangle}\ \textsf{lift} \qquad \frac{\langle \Psi; e_\alpha \rangle \mapsto \langle \Psi'; e \rangle}{\langle \Psi; \theta; \texttt{Const}(e_\alpha) \rangle \mapsto \langle \Psi'; \theta; \texttt{Const}(e) \rangle}\ \textsf{const} \qquad \frac{}{\langle \Psi; \theta, [t/x]; \texttt{Var}(x) \rangle \mapsto \langle \Psi; \theta, [t/x]; t \rangle}\ \textsf{var}$$

$$\frac{\langle \Psi; \theta; \vec{e_t} \rangle \mapsto \langle \Psi'; \theta'; \vec{e_t'} \rangle}{\langle \Psi; \theta; f(\vec{e_t}) \rangle \mapsto \langle \Psi'; \theta'; f(\vec{e_t'}) \rangle}\ \textsf{func} \qquad \frac{\langle \Psi; \theta; e_t \rangle \mapsto \langle \Psi'; \theta'; e_t' \rangle}{\langle \Psi; \theta; e_t, \vec{e_t} \rangle \mapsto \langle \Psi'; \theta'; e_t', \vec{e_t} \rangle}\ \textsf{seqHead} \qquad \frac{\langle \Psi; \theta; \vec{e_t} \rangle \mapsto \langle \Psi'; \theta'; \vec{e_t'} \rangle}{\langle \Psi; \theta; n, \vec{e_t} \rangle \mapsto \langle \Psi'; \theta'; n, \vec{e_t'} \rangle}\ \textsf{seqTail}$$

$$\frac{\langle \Psi; \theta; e_t \rangle \mapsto \langle \Psi'; \theta'; e_t' \rangle}{\langle \Psi; \theta; e_t\ \texttt{apply}\ \phi \rangle \mapsto \langle \Psi'; \theta'; e_t'\ \texttt{apply}\ \phi \rangle}\ \textsf{applyExp} \qquad \frac{}{\langle \Psi; \theta; n\ \texttt{apply}\ \phi \rangle \mapsto \langle \Psi; \theta; n\phi \rangle}\ \textsf{applyEnd} \qquad \frac{\langle \Psi; \theta; e_t \rangle \mapsto \langle \Psi'; \theta'; e_t' \rangle}{\langle \Psi; \theta; e_t\ \texttt{unify}\ \{\vec{u}\} \rangle \mapsto \langle \Psi'; \theta'; e_t'\ \texttt{unify}\ \{\vec{u}\} \rangle}\ \textsf{uExp}$$

$$\frac{mgu(n,t) = \bot}{\langle \Psi; \theta; n\ \texttt{unify}\ \{t\ \texttt{withMgu}\ \phi\ \texttt{=>}\ e,\ \vec{u}\} \rangle \mapsto \langle \Psi; \theta; n\ \texttt{unify}\ \{\vec{u}\} \rangle}\ \textsf{uPure1} \qquad \frac{mgu(n,t) = \phi}{\langle \Psi; \theta; n\ \texttt{unify}\ \{t\ \texttt{withMgu}\ \phi\ \texttt{=>}\ e,\ \vec{u}\} \rangle \mapsto \langle \Psi; \theta; e \rangle}\ \textsf{uPure2}$$

$$\frac{mgu(n,t) = \bot}{\langle \Psi; \theta; n\ \texttt{unify}\ \{t\ \texttt{>=>}\ e,\ \vec{u}\} \rangle \mapsto \langle \Psi; \theta; n\ \texttt{unify}\ \{\vec{u}\} \rangle}\ \textsf{uMut1} \qquad \frac{mgu(n,t) = \phi}{\langle \Psi; \theta; n\ \texttt{unify}\ \{t\ \texttt{>=>}\ e,\ \vec{u}\} \rangle \mapsto \langle \Psi; \theta \circ \phi; e \rangle}\ \textsf{uMut2}$$

Figure 4: Unification on Scala: Abstract Semantics

We call them *term expressions*. Terms are constants $\texttt{Const}(e_\alpha)$, logical variables $\texttt{Var}(x)$ or function applications $f(\vec{e_t})$. Scala expressions e_α are agnostic to our unification library, terms t are primitive expressions, while the expression e_t `apply` θ applies substitution θ to the result of e_t. Our main focus is on the unification construct e_t `unify` $\{\vec{u}\}$, where the *subject* (resultant term of e_t) is unified with cases expressed in $\vec{u}$. We have two forms of unification cases, shown in Section 1: t `withMgu` θ `=>` e expresses the pure (immutable) unification with t resulting in mgu θ and continuing with the body expression e with no side-effects on t (instantiations has to be done explicitly by applying θ on t, i.e., t `apply` θ). On the other hand, t `>=>` e denotes mutable unification with t: e is executed with mgu θ implicitly applied (as side-effects) to logical variables appearing in t and the unification subject.

Figure 4 defines the abstract semantics of the `unify` construct by means of state transitions of the form $\langle \Psi; \theta; e \rangle \mapsto \langle \Psi'; \theta'; e' \rangle$, called *unification evaluation*. Here, Ψ represents the abstract state of the Scala runtime, θ keeps track of the mutable changes imposed by the use of mutable unify cases, and e is the current expression being evaluated. $\langle \Psi'; \theta'; e' \rangle$ represents the resultant state. Expressions in normal form are denoted by n and the abstract evaluation of a standard expression e_α is written $\langle \Psi; e_\alpha \rangle \mapsto \langle \Psi'; e \rangle$. Such abstract evaluations are lifted into a unification evaluation by the (lift) rule. Note that the resultant state is of the form e, meaning that it may be a unification expression (a term, `unify` or `apply` expression). Rules (const), (var) and (func), together with auxiliary rules (seqHead) and (seqTail), inductively define the evaluation of terms in a standard way. The rule (applyExp) evaluates expressions e_t `apply` θ when e_t is not in normal form, while (applyEnd) defines the cases when it is. Similarly, the rule (uExp) defines the evaluation of a `unify` construct where its unification term subject is not in normal form, while all other cases — (uPure1/2) and (uMut1/2) — require otherwise. Rules (uPure1/2) handle the cases where the topmost unification case to attempt is a pure unify case statement t `withMgu` θ `=>` e: (uPure1) deals with the case where this unification attempt fails (i.e., $mgu(n,t) = \bot$) during which the remaining alternatives are attempted next, while

```
// type Subst = Map[LogVar[Any],Term[Any]]

//  Definitions of Terms
abstract class Term[A] {
     def mgu(other: Term[A]): Option[Subst]
     def apply(theta: Subst): Term[A]
     def unify[B](cases: UnifCase[A,B]*): B
     def withMgu[B](body: Subst => B): PureUnifCase[A,B]
     def >=>[B](body: => B): ImmUnifCase[A,B]
}
class LogVar[A] extends Term[A]
case class Const[A](n:A) extends Term[A]
case class Func[A](sym: String, args: List[Term[Any]]) extends Term[A]

//  Unify cases:  t withMgu θ => e and t >=> e
abstract class UnifCase[A,B](pat: Term[A])
class PureUnifCase[A,B](pat: Term[A], body: Subst => B) extends UnifCase[A,B](pat)
class MutUnifCase[A,B](pat: Term[A], body: => B) extends UnifCase[A,B](pat)

//  Unification Pattern Extractor
class Unif[A](pat: Term[A]) {
     def unapply(t: Term[A]): Option[Subst] = t mgu pat
}
```

Figure 5: Class Declarations of the Unification Library

(uPure2) defines the successful case (i.e., $mgu(n,t) = \theta$), in which the body expression e is executed next, with the implicit assignment of θ with the result of $mgu(n,t)$. Rules (uMut1/2) do the same for mutable unify cases: (uMut2) is similar to (uPure2) except that it additionally composes the substitution state θ with the resultant substitution ϕ of the unification of n and t. This models logical variable instantiation as side-effects of the case statement. Note that n is guaranteed not to contain any logical variables that appear in θ, since the corresponding (var) rule would have been applied to obtain the normal form term n itself. Hence the composition '$\theta \circ \phi$' is always well-defined.

3 Implementation

We now describe our implementation of the unification library in Scala. For brevity, we discuss only the public interfaces and combinator operations of this library. The full code is open-source and available for download at https://github.com/sllam/unifscala.

Figure 5 shows the class declarations of our unification library. Substitutions are implemented as maps from logical variables to terms and aliased as the type `Subst` for convenience. Terms are represented by the class `Term[A]`. We refer to polymorphic type `A` as the base type, and require that it is not a `Term` type itself. Terms are extended by three subclasses, namely `LogVar[A]` that represents logical variables, `Const[A]` that represents a constant value and `Func[A]` that represents a function application, where `A` is some base type. Constants and function applications are declared as 'case' classes and hence are treated as algebraic datatypes for convenient pattern matching. Logical variables are omitted from this however, since they are uniquely identified as references, rather than by their structure. Terms support a number of operations: `mgu(t)` unifies the current term with `t` and returns their most general unifier, if it exists. This method corresponds to the meta-operation $mgu(t,t')$ used in Section 2 and encapsulates the actual implementation of first-order term unification. The method `apply(`θ`)` simply applies the substitution θ onto the current term and returns the resultant term of this application. The method `unify` implements the control statement of the same name and takes a sequence of unification cases of type `UnifCase[A,B]`, where `A` is the base type of the term subject and `B` the return

value of the statement. Unification cases are constructed by two methods, namely `withMgu` and `>=>`, respectively corresponding to the syntax entities introduced in Figure 3. A term t calling `withMgu` takes a higher-order function θ `=>` e (of type `Subst => B`) and constructs the unification case t `withMgu` θ `=>` e. Similarly, t calling `>=>` takes a nullary higher-order function e (of type `=> B`) and constructs the unification case t `>=>` e. To support mutable side-effects, logical variables double as containers that can be instantiated with values. We omit these code fragments since their implementations are fairly standard.

We integrate our unification library into Scala's pattern matching framework by means of the extractor class `Unif`. Scala's extensible pattern matching framework is best described by the following classic example:

```
1  object Twice {
2    def unapply(x: Int): Option[Int] = if(x%2==0) Some(x/2) else None
3    def test(x: Int) {
4      x match {
5        case Twice(y) => println(x + "is even and twice " + y)
6        case _ => println(x + " is odd") } }
7  }
```

Here, `Twice` is declared as an object with an `unapply` method that takes an integer `x` and returns the value of half `x` only if `x` is even. This `unapply` method is implicitly called during pattern matching (in the illustrative method `test`) to extract the corresponding value `y`. As shown in Figure 5, we declare `Unif` as a class of extractors: given a term pattern `pat`, `Unif(pat)` constructs a unification pattern extractor for `pat`. The `unapply` method of this class of extractors simply takes a term `t` and attempts to unify `t` with `pat` to produce an mgu (i.e., `t mgu pat`). Looking back to Figure 2, unification pattern extractors for patterns `Const(4)` and `F(y,Const(5))` were defined in this manner and used to extract most general unifiers.

4 Conclusion

We have developed a lightweight unification library in Scala that allows the programmer to express first-order term unification natively using either Scala's pattern matching framework or with combinators that resemble Scala's `match` statement. With this unification library as the core, we intend to build more sophisticated programming constructs, for instance backtracking statements that integrate with unification and logical reasoning frameworks. Our work here contributes a step towards easier development of high-level declarative domain specific languages (e.g., [2, 3]) that rely on unification.

References

[1] P. Haller and T. van Cutsem. Implementing Joins Using Extensible Pattern Matching. In *COORDINATION'08*, pages 135–152. Springer LNCS 5052.

[2] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *POPL'12*, pages 151–164.

[3] E. S.L. Lam, I. Cervesato, and N. Fatima Haque. Comingle: Distributed Logic Programming for Decentralized Mobile Ensembles. In *COORDINATION'15*, pages 51–66. Springer LNCS 9037.

[4] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[5] F. Raiser. Scala-Logic Library. https://github.com/FrankRaiser/scala-logic.