Scalog Dokumentation

Florian Dobener
Betreuer: Prof. Dr. H. P. Gumm

29. August 2011

Inhaltsverzeichnis

1 Über Scalog

1.1 Die Idee hinter Scalog

Scalog soll es ermöglichen, Konstrukte der logischen Programmiersprache Prolog direkt im Quelltext von Scala verwenden zu können. Dabei soll eine einfache Scala nahe der Syntax das Arbeiten erleichtern. Die aktuelle Implementation nimmt Prolog Definitionen über einen eigens dafür definierten Codeblock an und stellt diese Funktionen als einfache Scala Funktionen dar, die wie jede andere Funktion in Scala verwendet werden können.

1.2 Scalog verwenden

Prolog Codeblöcke werden über das Schlüsselwort $\%prolog[\dots]\{\dots\}$ gestartet. In den eckigen Klammern werden die Funktionen, die von Prolog nach Scala exportiert werden sollen definiert. Dieser Aufruf hat die Form $< ScalaFunktion >=>< PrologFunktion >$, wobei $< ScalaFunktion >$ eine gewöhnliche Scala Funktionsdeklaration ist (ohne def), genauso wie $< PrologFunktion >$ ein gewöhnlicher Prolog Funktionsaufruf sein muss. Dabei gibt es allerdings einige Konventionen zu beachten:

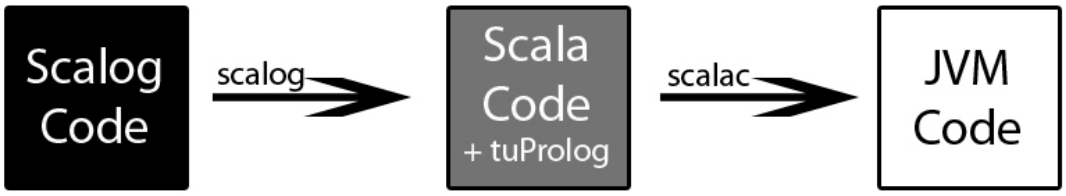
- Die Rückgabewerte der Scala Funktion werden als Tupel angegeben. Außerdem werden diese in Options umgewandelt, man erhält also immer einen Rückgabewert vom Typ Option[...] (siehe dazu die Beispiele in Sektion 4). Insbesondere muss auch ein einzelner Rückgabewert von Klammern umschlossen werden. Diese Klammerung soll aber noch beseitigt werden (⇒ Siehe Sektion 5).
- Mehrere Funktionen werden durch Kommata getrennt.
- Die Parameter der Scala Funktion und der Prolog Funktion müssen den gleichen Namen und die gleiche Anzahl haben. Außerdem dürfen die freien Variablen des Prolog Aufrufs nicht in größerer Anzahl als die Rückgabewerte der Scala Funktion vorkommen.
- Als freie Prolog Variablen werden alle Variablen erkannt, die nicht als Scala Parameter definiert wurden. Insbesondere auch kleingeschriebene Atome werden in dieser Version noch als Variablen erkannt (⇒ Siehe Sektion 5).

Nach der Bekanntmachung der Funktionen folgt der Prolog Body in geschweiften Klammern. Dort kann die gewöhnliche Prolog Syntax genutzt werden. Da dies direkt an tuProlog weiter gegeben wird, muss für eventuelle Einschränkungen die tuProlog Dokumentation zu Rate gezogen werden.

2 Implementierung

2.1 Grundidee

Scalog verwendet einen Präparser, d.h., dass der Code einer Datei, die Scalog Code enthält, durch den Parser in einen gewöhnlichen Scala Code (mit tuProlog API Benutzung) übersetzt wird. Folgendes Schaubild zeigt die Abfolge:



Dabei wird der Inhalt des Prolog Blocks direkt an die tuProlog API weiter gegeben, um von dort geparst zu werden. Die Funktionsexporte in eckigen Klammern werden in eine Scala Funktion übersetzt, die über ein tuProlog Interface eine direkte Anfrage an die in tuProlog definierte Theorie stellt.

2.2 Aufbau des Quelltextes

Die Hauptklasse ScalogPreParser spaltet sich in drei größere Bereiche auf. Die Klasse erbt von JavaTokenParsers, die die Parserfunktionalität bereit stellt.

2.2.1 Predefinitions

Der erste Teil des Quelltextes dient zur Definition von Systemfunktionen, die für die Verwendung der Prolog Funktionen notwendig sind (z.B. implizite Konversationen etc.). Dieser Teil wird unverändert in die Scala Datei übernommen. Hier muss also ein valider Scala Code ohne Prolog Definitionen vorhanden sein.

2.2.2 Der Parser

Der Parser wurde mit Scalas Parsing Combinators implementiert. Diese erlauben es, eine kontextfreie oder reguläre Grammatik ohne weitere Änderungen direkt in das Programm zu implementieren (für die Definition der Grammatik siehe Abschnitt 3). Der Übersetzer arbeitet hauptsächlich mit String Rückgabewerten, sodass diese direkt in die Scala Datei geschrieben werden können ohne das Ganze noch einmal umzuwandeln.

2.2.3 Parserfunktionen

Die Parserfunktionen dienen dazu, die Ausgangselemente (wie sie aus der Datei heraus geparst werden) in das gewünschte Format zu konvertieren. In dem beschrieben Fall ist das das String Format, damit die Daten ohne Umwandlung in eine Datei geschrieben werden können. Nähere Informationen zu den Funktionen gibt es in den Scaladocs.

2.2.4 Das Scalog Objekt

Das Scalog Objekt dient lediglich dazu einen Datenstrom zu nehmen, ihn vom Scalog-PreParser umwandeln zu lassen und diesen dann wieder in eine Datei zu schreiben.

2.3 tuProlog

Die Bibliotheken des tuProlog Projektes (`http://alice.unibo.it/xwiki/bin/view/Tuprolog/`) stellen einen Grundpfeiler von Scalog dar. Sämtliche Prolog Ausdrücke werden damit geparst und verarbeitet. Deshalb wird hier in verkürzter Form auf die verwendeten Konstrukte der tuProlog API eingegangen. Für ein umfassenderes Verständnis ist die tuProlog Dokumentation zu Rate zu ziehen. Im Scalog Projekt wird lediglich eine Prolog Engine instanziert und die Theorie aus dem Prolog Block als String an diese Engine übergeben. Somit ist man dazu in der Lage, die jeweiligen Funktionen abzufragen. Folgende Funktionen und Klassen wurden für das Projekt verwendet:

- Die Klasse **Prolog**, die sämtliche nötigen Funktionen bereit stellt. Sie kann ohne Parameter instanziert werden.
- Die Funktion **setTheory** der Prolog Klasse lädt eine Theorie als Stream oder String. Hier wird ein String übergeben, der aus dem Prolog Block extrahiert wurde.
- Die Funktion **solve** der Prolog Klasse löst einen gegebenen Prolog Ausdruck und gibt den Type SolveInfo zurück, der ebenfalls in den tuProlog Bibliotheken implementiert ist.
- Die Klasse **SolveInfo** beinhaltet alle Informationen einer solve Anfrage, wobei in Scalog nur zwei Funktionen dieser Klasse benötigt werden. Zum einen die Funktion **isSuccess**, die anzeigt ob der Ausdruck gelöst werden konnte und zum anderen die Funktion **getVarValue**, die einen String als Parameter verwendet und deren Inhalt als tuProlog Term ausgibt.

3 Grammatik

3.1 Definition

```
scalaFile    ::=  [ scalaExpr ] [ prologDef ] [ scalaExpr ] ;
prologDef    ::=  "%prolog" prologFunDef "{" prologBody "}" ;
prologFunDef ::=  "[" funcs "]" ;
funcs        ::=  func { "," func } ;
func         ::=  id "(" funArgs ")" ":" "(" varTypes ")"
                  "=>" id "(" ids ")" ;
funArgs      ::=  funArg { "," funArg } ;
varTypes     ::=  varType { "," varType } ;
funArg       ::=  id ":" varType;
varType      ::=  "Int"
                | "Double"
                | "String"
                | ( "List" | "PrologList" ) "[" varType "]" ;
ids          ::=  id { "," id } ;
```

Die Sprache enthält vier weitere Konstrukte: scalaExpr, prologBody, stringLiteral und id, welche durch reguläre Ausdrücken realisiert wurden. Hier ihre Definition:

```
scalaExpr     ::=  [^%]*
prologBody    ::=  [^}]*
stringLiteral ::=  StringRegex aus der Klasse JavaTokenParsers.
id            ::=  [a-zA-Z]\w*
```

Die eigentliche Definition der Prolog Konstrukte lautet in der Regel prologDef, da scalaFile lediglich dazu dient, den Quelltext in verschiedene Bereiche zu untergliedern.

4 Beispiele

4.1 Eine Funktion von Prolog in Scala verwenden

Wir wollen eine einfache Funktion in Prolog implementieren und diese in Scala verwenden. Zuerst wollen wir eine Funktion definieren, die eine Liste mit Strings umkehrt. Dazu definieren wir ein Objekt mit einem Prolog Block und schreiben unsere Definitionen hinein. Die Funktion *reverse* können wir durch den Export in eckigen Klammern nun in Scala verwenden.

```
// file: ReverseTest.scalog

object ReverseTest{
       %prolog[reverse(L1:List[String]):(List[String]) => reverse(L1,R)]{
              reverse(I,O) :- reverseAcc(I,[],O).
              reverseAcc([],X,X).
              reverseAcc([H|T],X,Y) :- reverseAcc(T,[H|X],Y).
```

```
        }

        def main(args:Array[String]){
                println(reverse(List("f","o","o","b","a","r")))
        }
}
```

Diese Funktion parst - mit dem Befehl **scala scalog.Scalog ReverseTest.scalog ReverseTest.scala** - in folgende Scala Datei:

```
// file: ReverseTest.scala
import alice.tuprolog._
import scala.util.parsing.combinator._

object ReverseTest{


        //Scalog Definition Part

        /*******************************************
        Hier befindet sich normalerweise der Predef Teil des Quelltextes.
        Dieser wurde zur besseren Lesbarkeit entfernt.
        *******************************************/


        engine.setTheory(new Theory("""reverse(I,O) :- reverseAcc(I,[],O).
                reverseAcc([],X,X).
                reverseAcc([H|T],X,Y) :- reverseAcc(T,[H|X],Y).
                """))

        def reverse(L1:PrologList[String]):(Option[PrologList[String]]) = {
                val result = engine.solve("reverse(" + L1 + " , R).")

                var r0:Option[PrologList[String]] = None
                if(result.isSuccess) r0 = Some(result.getVarValue("R").toString)


                (r0)
        }
        //End of Scalog Definition Part

        def main(args:Array[String]){
                println(reverse(List("f","o","o","b","a","r")))
        }
}
```

Besonders ist hier der Umstand zu beachten, dass die Funktion *reverseAcc* zwar im Prolog Block definiert ist, aber keine Scala Definition erzeugt wird. Dies rührt daher, dass Scalog lediglich alle Funktionen aus dem Export Statement übernimmt und ansons-

ten alle Definitionen im Prolog Block vernachlässigt. So kann man sich Hilfsmethoden generieren, die nach außen (in Scala) nicht sichtbar sind.

4.2 Mehrere Funktionen verwenden

Wir wollen nun versuchen, etwas kompliziertere Funktionen mit mehreren Parametern zu erhalten. Außerdem wollen wir dieses Mal zwei Funktionen gleichzeitig exportieren. Die erste Funktion, mit der man zwei Listen mit Strings konkatenieren kann, ist append. Die zweite Funktion pyth liefert uns ohne einen Eingabewert ein pythagoraeisches Tripel (also drei Int Rückgabewerte).

```
// file: AppCubicTest.scalog
object AppCubicTest{
	%prolog[append(L1:List[String],L2:List[String]):(List[String]) =>
	    append(L1,L2,R), cubic(L1:Int):(Int) => cubicRoot(L1,R1)]{

		append([],X,X).
		append([H|T],X,[H|Rest]) :- append(T,X,Rest).

		cubicRoot(1,1).
		cubicRoot(X,E) :- X > 0, cubicRootAcc(X,X,E).
		cubicRootAcc(X,K,K) :- X >= K*K*K, !.
		cubicRootAcc(X,K,E) :- K2 is K - 1, cubicRootAcc(X,K2,E).
	}

	def main(args:Array[String]){
		println(append(List("f","o","o"),List("b","a","r")))
		println(cubic(27))
	}
}
```

Durch den Aufruf **scala scalog.Scalog AppPythTest.scalog AppPythTest.scala** generieren wir wieder folgende Datei:

```
// file: AppCubicTest.scala
import alice.tuprolog._
import scala.util.parsing.combinator._

object AppCubicTest{


	//Scalog Definition Part

	/*******************************************
	Hier befindet sich normalerweise der Predef Teil des Quelltextes.
	Dieser wurde zur besseren Lesbarkeit entfernt.
	*******************************************/

	engine.setTheory(new Theory("""append([],X,X).
```

```
                append([H|T],X,[H|Rest]) :- append(T,X,Rest).

            cubicRoot(1,1).
                cubicRoot(X,E) :- X > 0, cubicRootAcc(X,X,E).
                cubicRootAcc(X,K,K) :- X >= K*K*K, !.
                cubicRootAcc(X,K,E) :- K2 is K - 1, cubicRootAcc(X,K2,E).
                """))

    def append(L1:PrologList[String],
        L2:PrologList[String]):(Option[PrologList[String]]) = {
            val result = engine.solve("append(" + L1 + " , " + L2 + " , R).")

            var r0:Option[PrologList[String]] = None
            if(result.isSuccess) r0 = Some(result.getVarValue("R").toString)


            (r0)
    }

    def cubic(L1:scala.Int):(Option[scala.Int]) = {
            val result = engine.solve("cubicRoot(" + L1 + " , R1).")

            var r0:Option[scala.Int] = None
            if(result.isSuccess) r0 = Some(result.getVarValue("R1").toString)


            (r0)
    }
    //End of Scalog Definition Part

    def main(args:Array[String]){
            println(append(List("f","o","o"),List("b","a","r")))
            println(cubic(27))
    }
}
```

4.3 Einsteins Rätsel

Nun wollen wir versuchen, ein Problem zu lösen, das sich in Scala schwierig lösen lässt aber für Prolog nicht sehr schwer zu implementieren ist: Einsteins berühmtes Rätsel (für die, die es nicht kennen: `http://www.adventure-yachting.de/Mathematik/einsteinraetsel.htm`). Wer besitzt den Fisch? Hier die Lösung:

```
// file: Einstein.scalog
object Einstein{
    %prolog[einstein():(String,String) => run(R1,R2)]{
            erstes(E,[E|_]).
            mittleres(M,[_,_,M,_,_]).
```

```
                links(A,B,[A|[B|_]]).
                links(A,B,[_|R]) :- links(A,B,R).

                neben(A,B,L) :- links(A,B,L);links(B,A,L).

                run(X,N) :-
                        X = [_,_,_,_,_],
                        member([rot,brite,_,_,_],X),
                        member([_,schwede,_,_,hund],X),
                        member([_,daene,tee,_,_],X),
                        links([gruen,_,_,_,_],[weiss,_,_,_,_],X),
                        member([gruen,_,kaffee,_,_],X),
                        member([_,_,_,pallmall,vogel],X),
                        mittleres([_,_,milch,_,_],X),
                        member([gelb,_,_,dunhill,_],X),
                        erstes([_,norweger,_,_,_],X),
                        neben([_,_,_,marlboro,_],[_,_,_,_,katze],X),
                        neben([_,_,_,_,pferd],[_,_,_,dunhill,_],X),
                        member([_,_,bier,winfield,_],X),
                        neben([_,norweger,_,_,_],[blau,_,_,_,_],X),
                        member([_,deutsche,_,rothmans,_],X),
                        neben([_,_,_,marlboro,_],[_,_,wasser,_,_],X),
                        member([_,N,_,_,fisch],X).
        }

        def main(args:Array[String]){
                val whoHasTheFish = einstein
                whoHasTheFish._2 match {
                        case Some(x) => println("Der " + x + " hat den Fisch.")
                        case None => println("Scheinbar hat niemand den Fisch.")
                }
                println("\nHaeuserbelegung: " + whoHasTheFish._1)
        }
}
```

Das Ganze wird wieder mit **scala scalog.Scalog Einstein.scalog Einstein.scala** übersetzt und liefert uns folgende Datei:

```
import alice.tuprolog._
import scala.util.parsing.combinator._

object Einstein{


        //Scalog Definition Part
```

```
/*******************************************
Hier befindet sich normalerweise der Predef Teil des Quelltextes.
Dieser wurde zur besseren Lesbarkeit entfernt.
*******************************************/

engine.setTheory(new Theory("""erstes(E,[E|_]).
        mittleres(M,[_,_,M,_,_]).

        links(A,B,[A|[B|_]]).
        links(A,B,[_|R]) :- links(A,B,R).

        neben(A,B,L) :- links(A,B,L);links(B,A,L).

        run(X,N) :-
                X = [_,_,_,_,_],
                member([rot,brite,_,_,_],X),
                member([_,schwede,_,_,hund],X),
                member([_,daene,tee,_,_],X),
                links([gruen,_,_,_,_],[weiss,_,_,_,_],X),
                member([gruen,_,kaffee,_,_],X),
                member([_,_,_,pallmall,vogel],X),
                mittleres([_,_,milch,_,_],X),
                member([gelb,_,_,dunhill,_],X),
                erstes([_,norweger,_,_,_],X),
                neben([_,_,_,marlboro,_],[_,_,_,_,katze],X),
                neben([_,_,_,_,pferd],[_,_,_,dunhill,_],X),
                member([_,_,bier,winfield,_],X),
                neben([_,norweger,_,_,_],[blau,_,_,_,_],X),
                member([_,deutsche,_,rothmans,_],X),
                neben([_,_,_,marlboro,_],[_,_,wasser,_,_],X),
                member([_,N,_,_,fisch],X).
        """))

def einstein():(Option[String], Option[String]) = {
        val result = engine.solve("run(R1, R2).")

        var r0:Option[String] = None
        if(result.isSuccess) r0 = Some(result.getVarValue("R1").toString)
        var r1:Option[String] = None
        if(result.isSuccess) r1 = Some(result.getVarValue("R2").toString)


        (r0, r1)
}
//End of Scalog Definition Part
```

```
        def main(args:Array[String]){
                val whoHasTheFish = einstein
                whoHasTheFish._2 match {
                        case Some(x) => println("Der " + x + " hat den Fisch.")
                        case None => println("Scheinbar hat niemand den Fisch.")
                }
                println("\nHaeuserbelegung: " + whoHasTheFish._1)
        }
}
```

5 Offene Probleme

5.1 Bugs und Probleme

- **Zulassen von mehreren Codeblöcken:** Im Moment ist es nur möglich, einen Prolog Block in den Quelltext einzufügen. Hierzu muss die scalaFile Regel der Grammatik editiert werden und insofern geändert werden, dass mehrere Codeblöcke möglich werden.

- **Verbesserung des regulären Ausdrucks scalaExpr:** Dieser wurde bisher so implementiert, dass der reguläre Ausdruck terminiert, wenn er auf ein %-Zeichen trifft. Allerdings könnte das Zeichen auch im Scala Quelltext vorkommen und der Compiler würde terminieren. Außerdem muss in diesem Fall das Codewort %prolog anstatt prolog zur Definition des Prolog Blocks verwendet werden.

- **Argumentprüfung:** Die Argumente der Scala Definitionen sollten gegen die Argumente der Prolog Funktion geprüft werden. Im Moment wird keine Rücksicht darauf genommen, ob die Variablen wirklich übereinstimmen. Im Moment wird dieser Fehler erst beim Kompilieren der Scala Datei ersichtlich, allerdings sollte dies von Scalog aufgefangen und in einer entsprechenden Fehlermeldung repräsentiert werden.

- **Rückgabewerte nicht als Tupel:** Bei einem Rückgabewert der nach Scala exportierten Funktion soll es auch möglich sein, diesen als gewöhnlichen Rückgabewert, nicht als Tupel, zu behandeln. Dazu muss die func Regel der Grammatik erweitert werden.

- **Berücksichtigung von mehreren Rückgabewerten:** Einer der größten Vorteile Prologs ist es, mehrere Ergebnisse mit nur einer Anfrage zu liefern. Die Implementation Prologs liefert aber jeweils nur die erste davon. Hier wäre es wünschenswert, ein Konstrukt zu finden, mit dessen Hilfe alle Werte abgefragt werden können - eventuell auch über Mehrfachaufrufe der Anfrage oder durch eine Rückgabe aller Werte als Liste (was bei unendlich vielen Ergebnissen jedoch schwierig wird).

5.2 Weitere Implementationsideen

- **Generische Typen:** Die Syntax rückt hier näher an Prolog Regeln heran. Bisher musste man die Parameter der Scala Funktionen statisch typisieren, weshalb es hier wünschenswert wäre, dass die generische Typisierung Prologs übernommen wird. Es muss allerdings ausgeschlossen werden, dass nur Prolog Typen übergeben werden können, aber keine selbst generierte Scala Klasse (bzw. müsste diese dann eine entsprechende toString Methode besitzen, sodass das Objekt in einen entsprechenden Ausdruck konvertiert wird).

- **Stärkere tuProlog Bindung:** Eine Verwendung der Parser Klasse aus tuProlog könnte den Übersetzungsprozess flexibler und stabiler machen. Außerdem könnte der Parser direkt in tuPrologs Term Konstrukte übersetzen, die eine schnellere und sichtbarere Umsetzung in den Scala Quelltext erfüllen würden.

- **Prolog-Standard zur Ein-/Ausgabe:** In Prolog ist es Konvention, dass Elemente die übergeben werden, und Elemente, die zurück gegeben werden, mit einem + und einem - gekennzeichnet werden. Diese Syntax wäre für die Prolog Funktionsdefinition wünschenswert, da sie die Interaktion verbessern und auch den Übersetzungsprozess typsicherer machen könnte.