MiniKanren

Implementierung in Scala

FELIX KECK, Universität Tübingen, Deutschland

In diesem Paper erläutere ich, wie ich die relationale Sprache MiniKanren, welche ursprünglich in Scheme geschrieben ist, in Scala implementiert habe. Dabei gehe ich insbesondere darauf ein, welche Datentypen dafür notwendig waren, wie sich die Kernfunktionen der beiden Implementierungen unterscheiden und welche Probleme beim Implementieren aufgetreten sind. MiniKanren ist im Buch "The Reasoned Schemer" von Daniel P. Friedman definiert, welches ich in der zweiten Auflage verwende, veröffentlicht 2018 von The MIT Press.

1 WARUM SCALA

Als Grundlage dient die Implementierung von MiniKanren in Scheme, einer funktionalen, dynamisch getypten Programmiersprache. Für meine Implementierung fiel die Wahl recht schnell auf Scala. Zum einen habe ich bereits einige Projekte in Scala implementiert und dadurch ausreichend Erfahrung in dieser Sprache. Zum anderen vereint Scala wie nur wenig andere Sprachen so viele Konzepte (imperative, funktionale, objekt-orientierte Programmierung) in einer Sprache, ohne dabei komplizierte oder unverständliche Syntax zu verwenden. Diese Vielfalt bietet den Vorteil, z.B. das Verhalten von Variablen in einer Klasse zu kapseln aber gleichzeitig Lambda-Ausdrücke eins zu eins übernehmen zu können. Der größte Unterschied zwischen den beiden Programmiersprachen ist jedoch, dass Scala im Gegensatz zu Scheme statisch getypt ist. Welche Schwierigkeiten sich dadurch ergeben, erläutere ich in Abschnitt 3.

2 DATENTYPEN

Im Folgenden stelle ich vor, wie ich die grundlegenden Datentypen von MiniKanren implementiert habe.

2.1 Terme

```
1  trait Term
2  case class Lit(v: String) extends Term
3  case object End extends Term
4  case class Pair(left: Term, right: Term) extends Term
5  case class Var(varName: String) extends Term {
6    var name: String = varName
7    val id: Int = varCounter
8    varCounter += 1
9  }
```

Die Eingabeparameter aller Relationen in MiniKanren sind Terme. Terme sind entweder Literale (z.B. `Lit("1")`), Paare von Termen, das leere Paar (`End`) oder Variablen (z.B. `Var("x")`). Um solche disjunkten Subtypen darzustellen, eignet sich die Komposition aus einem *Trait* für den abstrakten Basistyp mit mehreren *case classes* bzw. *case objects* für die konkreten Typen. Paare kodieren Listen, wobei auf der linken Seite ein beliebiges Element stehen kann, auf der rechten Seite muss jedoch immer entweder die Rest-Liste, also ein weiteres `Pair`, oder das `End`-Element stehen, um anzugeben, dass die Liste abgeschlossen ist. Die Kodierung der Liste bestehend aus den Elementen `1`, `2` und `3` als Term sieht wie folgt aus:
`Pair(Lit("1"), Pair(Lit("2"), Pair(Lit("3"), End)))`
Variablen bekommen im Konstruktor zwar einen Namen, intern bekommen sie aber noch eine Id

zugewiesen, die mittels eines globalen Counters immer hochgezählt wird, sobald eine neue Variable erstellt wird. Um zu vergleichen, ob zwei Variablen identisch sind, wird ausschließlich diese Id verglichen. Somit können problemlos mehrere Variablen mit dem selben Namen erstellt werden.

2.2 Streams

```
trait Stream[+T]
case object Empty extends Stream[Nothing]
case class Cons[T](elem: ()=>T, rst: ()=>Stream[T]) extends Stream[T]
case class Susp[T](func: () => Stream[T]) extends Stream[T]
```

Das zentrale Konzept zur Repräsentation von Lösungsschritten im Kern von Minikanren sind Streams. Streams verhalten sich im Prinzip wie normale Listen, jedoch mit ein paar Besonderheiten. Zum einen können Streams unendlich sein. Damit der Rest-Stream dieser rekursiven Datenstruktur nicht direkt ausgewertet wird, kapsle ich diesen in ein parameterloses Lambda, auch *Thunk* genannt. Das war notwendig, da Scala keine call-by-name Parameter in case-Klassen erlaubt. Zum anderen können Streams auch Suspensions sein oder enthalten. Eine Suspension ist selbst wieder eine Funktion, die keine Parameter hat, und, wenn ausgewertet, einen Stream vom selben Typ liefert. Suspensions waren der Grund, warum ich für Streams einen eigenen Datentyp angelegt habe, und nicht Scalas eingebaute `LazyLists` verwendet habe, welche, wie der Name schon sagt, Lazy Evaluation bereits unterstützen. Analog zu `End` bei Paaren, enden Streams immer mit `Empty`, dem leeren Stream, sofern sie nicht unendlich sind. Streams sind zwar generisch definiert, weshalb Streams von jedem beliebigen Typ erstellt werden können, also auch Streams von Integern oder Strings, bei der Implementierung von MiniKanren werden jedoch nur Streams von Substitutions verwendet.

2.3 Substitutions

```
type Subst = Map[Var, Term]
```

Bei der Auswertung von MiniKanren-Programmen werden nach und nach Assoziationen zwischen Variablen und Werten hergestellt. Diese Werte können beliebige Terme sein, also Literale, Paare, oder selbst wieder Variablen. Bei letzterem wird im Buch von *fusing* gesprochen, was ausdrückt, dass diese Variablen den selben Wert haben. Implementiert habe ich diesen Sachverhalt mit Maps, die eine Menge von Schlüsseln auf Werte abbilden und damit genau die gewünschte Aufgabe erfüllen. Die Typdefinition `Subst` stellt dabei lediglich einen Alias für `Map[Var, Term]` dar.

2.4 Goals

```
type Goal = Subst => Stream[Subst]
```

Der letzte wichtige Datentyp in MiniKanren sind Goals. Goals sind Funktionen, die eine Substitution erwarten und einen Stream von Substitutionen liefern. Ein Goal kann entweder Erfolg haben oder fehlschlagen. Meine Implementierung dieser beiden Goals ist analog zum Buch:

```
def succeed: Goal = {
  s: Subst => Stream.cons(s, Empty)
}

def fail: Goal = {
  _: Subst => Empty
}
```

3 VERGLEICH DER IMPLEMENTIERUNGEN

Auf die groben Unterschiede der beiden Host-Sprachen Scheme und Scala bin ich in bereits in Abschnitt 1 eingegangen. Welche Auswirkungen das auf meine Implementierung hatte und welche Schwierigkeiten dadurch aufgetreten sind, darum soll es in diesem Abschnitt gehen.

Als erstes ist festzuhalten, dass alle in Abschnitt 2 vorgestellten Datentypen in der Scheme Implementierung nicht notwendig sind, da durch die dynamische Typisierung keine Typannotationen existieren. Terme werden dort einfach durch Scheme Sprachfeatures ausgedrückt: Literale sind Scheme Literale (Integer, Symbole), Paare sind Scheme Listen und Variablen sind Scheme Variablen (diese werden intern durch Lambdas auf Vektoren kodiert). Ebenso gibt es auch keine Definition von Streams, auch hier werden einfach Scheme Listen verwendet. Die Einbindung von Suspensions ist ebenfalls dank der dynamischen Typen kein Problem. Substitutions in Scheme sind durch Listen von Paaren definiert. Meine Variante mit Maps hat hierbei den Vorteil, dass man nicht aus Versehen zwei Einträge mit dem selben Schlüssel anlegen kann. Für die Typdefinitionen war also in Scala Code notwendig, den es in Scheme gar nicht gibt. Auf den ersten Blick war auch nicht immer direkt ersichtlich, welchen Typ ein Parameter oder eine Funktion in Scheme hat. Dies musste man sich immer aus dem Kontext der Funktion herleiten. Meiner Meinung nach hat man durch die statische Typisierung dann aber auch Vorteile wie Compilerunterstützungen, bessere Verständlichkeit der Funktionen und implizite Dokumentation des Codes.

Ansonsten sind die meisten Implementierungen von Kernfunktionen wie z.B. `append-inf`, `reify-s` oder `walk*` identisch aufgebaut. Was in Scheme eine Aneinanderreihung von `cond`-Argumenten ist, löse ich in Scala mittels Pattern Matching, wie am Beispiel von `walk_star` zu sehen:

```
1  def walk_star(v: Term, s: Subst): Term = {
2    val walked = walk(v, s)
3    walked match {
4      case v: Var => v
5      case Pair(l, r) => Pair(walk_star(l, s), walk_star(r, s))
6      case _ => walked
7    }
8  }
```

Das hat den Vorteil, dass beim Matchen auf ein `Pair` direkt Variablen für die beiden Elemente von `Pair` vergeben werden können, und nicht später nochmal mit `car` und `cdr` auf die Liste zugegriffen werden muss. Hier also eine minimale Ersparnis an Code zugunsten von Scala.

Ebenfalls durch die dynamische Typisierung ist es in Scheme möglich, in den Funktionen `ext-s` und `unify` **`false`** anstelle einer Substitution zurückzugeben, falls die Berechnung fehlschlägt. In Scala wäre dafür der `Either`-Typ in der Form `Either[Subst, Boolean]` notwendig. Da **`false`** aber nur das Fehlschlagen der Berechnung symbolisiert, ist hierfür der Scala `Option`-Typ in der Form `Option[Subst]` als Return-Typ besser geeignet. Das selbe Prinzip verwende ich für die Funktionen `take-inf`, `run-goal` und `run`, die eine Ganzzahl erwarten, die angibt, wie viele Ergebnisse berechnet werden sollen. Sollen alle Ergebnisse berechnet werden, wird das in Scheme mit dem Parameter **`false`** angegeben. In Scala muss auf diesem `Option`-Parameter dann einmal gepatternmatched werden, um herauszufinden, ob nun ein Integer (`Some(n)`) vorliegt oder nicht (`None`).

Eine weitere Schwierigkeit war das Auftreten von *named lets* in den Funkionen `ifte` und `once`,

was als rekursiver Einstiegspunkt dient. Hierfür habe ich aber einfach jeweils eine geschachtelte Funktion `loop` definiert, die die notwendigen Parameter bekommt und von der eigentlichen Funktion entsprechend aufgerufen wird.

Letztendlich folgten noch die wichtigen Funkionen `disj`, `conj`, `run` und `conde`, die in Form von Scheme Makros geschrieben sind. Hier musste man eigentlich nur das `...`-Pattern in Scala varargs übersetzen, wofür einfach nur ein Stern an den Typ des entsprechende Parameters angehängt wird. Für beliebig viele Goals als Eingabeparameter einer Funktion würde man in einem Scheme Makro `(function-name g ...)` schreiben, in Scala hingegen **def** `function_name(g: Goal*)`. Werden in einem Makro wie z.B. bei `run` mehrere Pattern angegeben, damit beim Aufruf mit nur einer Variable nicht extra eine Liste erstellt werden muss, überlade ich in Scala einfach die `run`-Methode.

Die beiden Funktionen `fresh` und `call/fresh` verwende ich in meiner Implementierung gar nicht bzw. nur damit die Aufrufe bei mir identisch zu denen im Buch sind. In Scheme werden diese Funktionen dafür gebraucht, für jede Scheme Variable intern eine MiniKanren Variable zu erstellen. Das habe ich anders gelöst, indem ich immer, wenn eine Variable benötigt wird, direkt eine Instanz meiner Variablen-Klasse erstelle.

Bei der Implementierung von Relation, dem eigentlichen Sinn des MiniKanren-Kerns, trat erneut das Problem der Lazyness auf, da diese Goals liefern können, die wiederum unendliche Streams enthalten. Hier war der Trick, händisch eine *Eta-Expansion* durchzuführen, also das eigentliche Goal in einem Lambda zu kapseln und anschließend mit der Lambda-Variable aufzurufen. Das verhindert eine sofortige Auswertung des Goals und dadurch eine Endlos-Rekursion. Am Beispiel der `nullo`-Relation erkennt man diese Kapselung des eigentlichen Goals `==(End, x)` besonders gut:

```
1  def nullo(x: Term): Goal = {
2    s: Subst => ==(End, x)(s)
3  }
```

Bei der `appendo`-Relation war es zusätzlich noch notwendig, den rekursiven Aufruf in einer **lazy** Variable zwischenzuspeichern.

4 BENUTZUNG

Im Prinzip lässt sich meine Implementierung genauso wie die Scheme Implementierung benutzen, mit dem Unterschied, dass bei mir alle Parameter in Form von Termen bereitgestellt werden müssen. Nachfolgend ein Vergleich des selben `appendo`-Aufrufs mit den jeweiligen Ergebnissen.

Aufruf in Scheme:

```
1  (run* q (appendo '(1 2 3) q '(1 2 3 4 5 6)))
```

Ergebnis: `'((4 5 6))`

Aufruf in Scala:

```
1  val q = Var("q")
2  run_star(u, appendo(listToPairOfLit(1, 2, 3), u,
3    listToPairOfLit(1, 2, 3, 4, 5, 6)))
```

Ergebnis: `List(Pair(Lit("4"),Pair(Lit("5"),Pair(Lit("6"),End))))`

Es ist offensichtlich, dass sowohl der Aufruf, als auch das Ergebnis bei meiner Implementierung komplizierter ist. Für die Eingabe von Listen habe ich eine Hilfsmethode geschrieben, die eine gegebene Liste von Elementen in die geschachtelte Pair-Repräsentation umwandelt. Das Ergebnis könnte man sich aus Gründen der Lesbarkeit natürlich auch noch pretty-printen lassen, ich habe hier aber mit Absicht darauf verzichtet, um die interne Repräsentation darzustellen. Des Weiteren muss ich jede verwendete Variable vorher in einer Scala-Variable speichern, da sie ja immer Argument von `run` oder `fresh` und von mindestens einem Goal ist.