Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion

Prof. Dr. Michael Hanus, M.Sc. Finn Teegen, Sandra Dylus

Institut für Informatik

1. Übung zur Vorlesung „Praktikum zu Fortgeschrittene Programmierung“

Projekt
Abgabe am Freitag, 28. Februar 2020 - 09:00

In diesem Projekt soll unter Verwendung von Haskell als Programmiersprache ein Interpreter sowie eine interaktive Umgebung für eine vereinfachte Variante von Prolog programmiert werden. Diese Variante unterscheidet sich in den folgenden Punkten von der aus der Vorlesung bekannten.

- Der Cut-Operator (`!`) wird nicht unterstützt.
- Es gibt zunächst keine vordefinierten Prädikate.
- Prädikate können nicht infix notiert werden.
- Es werden keine Module unterstützt.

Barbeitungshinweise

Das Projekt besteht aus insgesamt dreizehn Aufgaben. Davon sind alle Aufgaben mit Ausnahme derer, die als Bonusaufgaben gekennzeichnet sind, verpflichtend und müssen zum Bestehen des Praktikums erfolgreich und vollständig bearbeitet werden. Das Bearbeiten der Bonusaufgaben dagegen ist freiwillig, gleicht das eventuelle Nichtbearbeiten einer oder mehrerer Pflichtaufgaben aber nicht aus. Da die Aufgaben jeweils aufeinander aufbauen, sollen sie nacheinander bearbeitet werden. Eine Abgabe der Aufgaben im iLearn ist nicht notwendig. Beachten Sie bei der Bearbeitung des Projekts auch die folgenden Hinweise.

- Lesen Sie alle Aufgaben sorgfältig durch und beachten Sie eventuell vorhandene Hinweise.
- Bearbeiten Sie jede Aufgabe (mit Ausnahme der Bonusaufgaben) in einem separaten Modul.
- Exportieren Sie jeweils nur die notwendigen Typ- und Datenkonstruktoren sowie Funktionen.
- Schreiben Sie leserlichen Code und halten Sie sich nach Möglichkeit an den Haskell Style Guide.
- Dokumentieren Sie Ihr Programm ausführlich unter Verwendung von Typsignaturen und Kommentaren.
- Achten Sie darauf, dass keine Warnungen ausgegeben werden, wenn Sie mit `-Wall` kompilieren.
- Modifizieren oder ergänzen Sie die vordefinierten Module nicht.
- Fragen Sie bei Unklarheiten oder Problemen rechtzeitig(!) nach.

Viel Spaß und Erfolg beim Bearbeiten der Aufgaben!

Aufgabe 1 - Pretty Printing

0 Punkte

Zunächst geht es um die Darstellung von Termen. Das vorgegebene Modul Type.hs enthält die dafür notwendigen Definitionen zur Repräsentation von Namen (`VarName` und `CombName`) und Termen (`Term`) sowie darüber hinaus von Regeln (`Rule`), Programmen (`Prog`) und Anfragen (`Goal`) und bildet die Grundlage für diese und alle weiteren Aufgaben.

1. Definieren Sie eine Typklasse `Pretty`, die eine Methode `pretty :: a -> String` beinhalten soll, um Datentypen schön auszugeben.
2. Geben Sie eine Instanz für den vordefinierten Datentyp `Term` an, wobei "schön" in diesem Fall bedeutet, dass Terme in gültiger Prolog-Syntax dargestellt werden sollen. Achten Sie dabei insbesondere auch auf die korrekte Darstellung von Listentermen, wie einige der folgenden Beispiele verdeutlichen.

```
ghci> pretty (Var "A")
"A"
ghci> pretty (Comb "true" [])
"true"
ghci> pretty (Comb "[]" [])
"[]"
ghci> pretty (Comb "f" [Var "B", Var "_", Comb "true" []])
"f(B, _, true)"
ghci> pretty (Comb "." [Comb "true" [], Comb "[]" []])
"[true]"
ghci> pretty (Comb "." [Comb "true" [], Comb "." [Comb "g" [Var "C"], Comb "[]" []]])
"[true, g(C)]"
ghci> pretty (Comb "." [Comb "1" [], Comb "." [Comb "2" [], Comb "." [Comb "3" [], Comb "[]" []]
"[1, 2, 3]"
ghci> pretty (Comb "." [Comb "true" [], Var "D"])
"[true|D]"
ghci> pretty (Comb "." [Var "E", Comb "h" [Var "F", Comb "i" [Var "G"]]])
"[E|h(F, i(G))]"
ghci> pretty (Comb "." [Comb "true" [], Comb "." [Comb "true" [], Comb "true" []]])
"[true, true|true]"
ghci> pretty (Comb "." [Comb "[]" [], Comb "[]" []])
"[[]]"
ghci> pretty (Comb "." [Comb "." [Comb "true" [], Comb "[]" []], Comb "[]" []])
"[[true]]"
ghci> pretty (Comb "." [Var "H"])
".(H)"
ghci> pretty (Comb "." [Var "I", Comb "true" [], Comb "j" [Var "J"]])
".(I, true, j(J))"
ghci> pretty (Comb "." [Var "K", Comb "." [Var "L", Var "M", Var "N", Var "O"]])
"[K|.(L, M, N, O)]"
```

Aufgabe 2 - Variablen

0 Punkte

Für den weiteren Verlauf ist es relevant, vorhandene Variablennamen ermitteln sowie frische Variablennamen erzeugen zu können.

1. Definieren Sie eine Typklasse `Vars`, die eine Methode `allVars :: a -> [VarName]` enthält. Diese Methode soll alle in einem Datentyp enthaltenen Variablen zurückgeben. Geben Sie Instanzen dieser Typklasse für die vordefinierten Datentypen `Term`, `Rule`, `Prog` und `Goal` an.
2. Definieren Sie eine Funktion `freshVars :: [VarName]`, die eine unendliche Liste in Prolog gültiger Variablennamen zurückgibt.

Aufgabe 3 - Substitutionen

0 Punkte

Im nächsten Schritt sollen Substitutionen, d.h. Zuordnungen von Variablen zu Termen, als abstrakter Datentyp (ADT) realisiert werden.

1. Definieren Sie einen Datentyp `Subst` zur Repräsentation von Substitutionen, dessen Konstruktor(en) Sie explizit nicht exportieren.
2. Definieren Sie dann zwei Funktionen `empty :: Subst` und `single :: VarName -> Term -> Subst` zum Erstellen einer leeren Substitution bzw. einer Substitution, die lediglich eine einzelne Variable auf einen Term abbildet.
3. Implementieren Sie eine Funktion `apply :: Subst -> Term -> Term`, die eine Substitution auf einen Term anwendet.
4. Implementieren Sie außerdem eine Funktion `compose :: Subst -> Subst -> Subst`, die zwei Substitionen so miteinander komponiert, dass für alle Terme `t` und Substitutionen `s1` und `s2` stets die Eigenschaft `apply (compose s2 s1) t == apply s2 (apply s1 t)` erfüllt ist (Achten Sie auf die Reihenfolge!).
5. Implementieren Sie weiterhin eine Funktion `restrictTo :: [VarName] -> Subst -> Subst`, die eine Substitution auf eine gegebene Variablenmenge einschränkt.
6. Geben Sie für den Datentyp `Subst` eine Instanz der Typklasse `Pretty` an, um Substitutionen schön darzustellen. Solch eine Darstellung von Substitutionen kann bspw. wie folgt aussehen.

```
ghci> pretty empty
"{}"
ghci> pretty (compose (single "A" (Var "B")) (single "A" (Var "C")))
"{A -> C}"
ghci> pretty (compose (single "D" (Var "E")) (single "F" (Comb "f" [Var "D", Comb "true" []])))
"{F -> f(E, true), D -> E}"
ghci> pretty (compose (single "G" (Var "H")) (single "I" (Var "J")))
"{I -> J, G -> H}"
```

7. Geben Sie für den Datentyp `Subst` schließlich auch eine Instanz der Typklasse `Vars` an.

Aufgabe 4 - Unifikation

0 Punkte

Aufbauend auf Substitutionen soll nun die zentrale Operation der Logikprogrammierung realisiert werden: die Unifikation.

1. Definieren Sie als erstes eine Funktion `ds :: Term -> Term -> Maybe (Term, Term)`, die die Unstimmigkeitsmenge zweier Terme berechnet und sie als Paar zurückgibt. Sollte die Unstimmigkeitsmenge leer sein, soll die Funktion stattdessen `Nothing` zurückgeben. Überlegen Sie sich auch, wie Sie dabei mit anonymen Variablen (`_`) umgehen.
2. Definieren Sie anschließend eine Funktion `unify :: Term -> Term -> Maybe Subst`, die aufbauend auf der Funktion `ds` den allgemeinsten Unifikator für zwei Terme bestimmt, sofern die beiden Terme unifizierbar sind. Ansonsten soll die Funktion `Nothing` zurückgeben.

Aufgabe 5 - Umbenennung

0 Punkte

Als Vorbereitung zur SLD-Resolution sollen jetzt Varianten der im Programm vorhandenen Regeln erzeugt werden. Definieren Sie hierzu eine Funktion `rename` passenden Typs, die die Variablen einer Regel umbenennt. Berücksichtigen Sie dabei erneut anonyme Variablen (`_`) in geeigneter Art und Weise.

Aufgabe 6 - SLD-Resolution

0 Punkte

Zur Auswertung von Anfragen wird eine geeignete Repräsentation von SLD-Bäumen benötigt.

1. Definieren Sie einen Datentyp `SLDTree`, der SLD-Bäume repräsentiert. Dabei sollen die Kanten der SLD-Bäume mit dem jeweiligen Unifikator beschriftet sein, der in dem entsprechenden Resolutionsschritt berechnet wurde.

   Hinweis: Es kann sinnvoll sein, zu Debugging-Zwecken eine `Pretty`-Instanz für SLD-Bäume zu definieren.

2. Definieren Sie im Anschluss eine Funktion `sld :: Prog -> Goal -> SLDTree`, die den SLD-Baum zu einem Programm und einer Anfrage konstruiert.

   Hinweis: Die Konstruktion des SLD-Baums soll auf Grundlage der Selektionsstrategie FIRST erfolgen, d.h. es soll immer das am weitesten links stehende Literal zum Beweisen ausgewählt werden.

Ein SLD-Baum kann dann auf verschiedene Weisen durchlaufen werden, um alle Lösungen für eine Anfrage zu finden. Eine Lösung ist durch die Belegung der freien Variablen der ursprünglichen Anfrage gegeben. Suchstrategien können mithilfe des Typs

```
1 type Strategy = SLDTree -> [Subst]
```

ausgedrückt werden.

1. Definieren Sie eine Funktion `dfs :: Strategy`, die einen SLD-Baum mittels Tiefensuche durchläuft und dabei alle Lösungen zurückgibt.
2. Definieren Sie außerdem eine Funktion `bfs :: Strategy`, die einen SLD-Baum mittels Breitensuche durchläuft und dabei ebenfalls alle Lösungen zurückgibt.
3. Implementieren Sie schließlich eine Funktion `solve :: Strategy -> Prog -> Goal -> [Subst]`, die unter Verwendung einer gegebenen Suchstrategie zu einem Programm und einer Anfrage alle Lösungen berechnet. Dabei sollen sich die Antwortsubstitutionen auf Belegungen für die freie, nicht-anonymen Variablen der Anfrage beschränken.

Aufgabe 7 - Interaktive Umgebung

0 Punkte

Nun fehlt noch die interaktive Umgebung, die Sie in Form einer REPL umsetzen sollen. Beachten Sie dabei die folgenden Punkte.

- Die interaktive Umgebung soll solange aktiv sein, bis sie vom Benutzer beendet wird.
- Der Benutzer soll ein Programm laden können, das solange geladen bleibt, bis ein neues geladen wird.
- Es sollen Anfragen gestellt werden können, deren Lösungen nacheinander angefordert werden können.
- Der Benutzer soll zwischen Anfragen jederzeit die Suchstrategie wechseln können.
- Es soll eine Hilfe angezeigt werden können, die alle Funktionen der interaktiven Umgebung auflistet.
- Die interaktive Umgebung soll robust gegenüber Fehlern sein und stets sinnvolle Rückmeldungen ausgeben.

Nachfolgend ist die Ausgabe eines beispielhaften Sitzungsablaufs gegeben, an dem Sie sich orientieren können.

```
Welcome!
Type ":h" for help.
?- :h
Commands available from the prompt:
  <goal>      Solves/proves the specified goal.
  :h          Shows this help message.
  :l <file>   Loads the specified file.
```

```
  :q          Exits the interactive environment.
  :s <strat>  Sets the specified search strategy
              where <strat> is one of 'dfs', 'bfs', or 'iddfs'.
?- :l examples/append.pl
Loaded.
?- :s bfs
Strategy set to breadth-first search.
?- append(A, [B|_], [1, 2]).
{A -> [], B -> 1} ;
{A -> [1], B -> 2} ;
No more solutions.
?- :q
```

Hinweis: Das Modul Parser.hs stellt zum Einlesen von Programmen und Anfragen die Typklasse `Parse` mit der Methode `parse :: String -> Either String a` sowie Instanzen für die vordefinierten Datentypen `Prog` und `Goal` bereit. Der Rückgabewert der Methode ist entweder eine Fehlermeldung oder der erfolgreich eingelesene Wert. Darüber hinaus ist eine überladene Funktion `parseFile :: Parse a => String -> IO (Either String a)` enthalten, die einen Dateipfad erwartet und eine Fehlermeldung oder den erfolgreich eingelesenen Dateinhalt in der `IO`-Monade zurückgibt.

Aufgabe 8 - Testen

0 Punkte

Abschließend soll die korrekte Funktionalität des Interpreters sichergestellt werden, indem Sie Ihre Implementierung ausführlich unter Verwendung geeigneter Beispiele testen.

Hinweis: Das Archiv examples.zip enthält einige Beispielprogramme, die als Grundlage dienen können.

Zusatzaufgabe 9 - Prädikate höherer Ordnung

0 Punkte

Im Rahmen der Vorlesung wurden u.a. Prädikate höherer Ordnung behandelt: Mittels der Familie vordefinierter `call`-Prädikate können Prädikate oder Prädikataufrufe als Argument an andere Prädikate übergeben und dann als Anfrage interpretiert werden. Erweitern Sie Ihre Implementierung entsprechend um die Unterstützung für Prädikate höherer Ordnung.

Zusatzaufgabe 10 - Negation

0 Punkte

Aus der Vorlesung ist ebenfalls die Negation als Fehlschlag bekannt. Erweitern Sie Ihren Interpreter um das vordefinierte, einstellige Prädikat `\+`. Berücksichtigen Sie bei der Implementierung auch die verschiedenen Suchstrategien.

Hinweis: Die Berücksichtigung verschiedener Suchstrategien macht es ggf. erforderlich, die zuvor definierte Funktion `sld` um einen Parameter für die Suchstrategie zu erweitern.

Zusatzaufgabe 11 - Kapselung

0 Punkte

Bisher fehlt noch die Unterstützung für die Kapselung von Nichtdeterminismus. Ergänzen Sie Ihren Interpreter hierfür um das vordefinierte, dreistellige Prädikat `findall` und stellen Sie alle Aspekte seines Originalverhaltens nach. Untersuchen Sie insbesondere, wie sich das `findall`-Prädikat anderer Prolog-Implementierungen verhält, wenn das erste Argument freie, nicht im zweiten Argument vorkommende Variablen enthält. Berücksichtigen Sie auch wieder die verschiedenen Suchstrategien.

Hinweis: Die Berücksichtigung verschiedener Suchstrategien macht es ggf. erforderlich, die zuvor definierte Funktion `sld` um einen Parameter für die Suchstrategie zu erweitern.

Zusatzaufgabe 12 - Arithmetik

0 Punkte

Zu guter Letzt soll der Interpreter um die Unterstützung für das Rechnen mit arithmetischen Ausdrücken ergänzt werden, wie es in anderen Prolog-Implementierungen möglich ist.

1. Implementieren Sie eine Funktion `eval :: Term -> Maybe Integer`, die den Wert eines arithmetischen Ausdrucks berechnet. Falls der arithmetische Ausdruck nicht wohlgeformt ist oder noch Variablen enthalten sollte, soll die Funktion `Nothing` zurückgeben.

   Hinweis: Im Rahmen dieser Aufgabenstellung bestehen arithmetische Ausdrücke ausschließlich aus ganzen Zahlen sowie den binären Funktoren `+`, `-`, `*`, `div` sowie `mod`. Der Interpreter muss also keine Gleitkommazahlen unterstützen.

2. Erweitern Sie den Interpreter auf Basis der Funktion `eval` um das vordefinierte, zweistellige Prädikat `is`.

3. Erweitern Sie den Interpreter auf Basis der Funktion `eval` zudem um die vordefinierten Vergleichsprädikate `=:=`, `=\=`, `<`, `>`, `>=` sowie `=<`.

Zusatzaufgabe 13 - Iterative Tiefensuche

0 Punkte

Die iterative Tiefensuche vereint die Vollständigkeit der Breitensuche und den geringen Speicherverbrauch der Tiefensuche. Definieren Sie daher eine Funktion `iddfs :: Strategy`, die einen SLD-Baum mittels iterativer Tiefensuche durchläuft und dabei alle Lösungen zurückgibt. Achten Sie dabei darauf, dass Sie bereits gefundene Lösungen nicht mehrfach in die Ergebnisliste aufnehmen und dass die iterative Tiefensuche für endliche SLD-Bäume terminiert.