Vom Matching zur Unifikation im Tupelraum

Wilhelm Hasselbring

Universität Dortmund, LS Informatik 10 (Software-Technologie)
D-44221 Dortmund, willi@ls10.informatik.uni-dortmund.de

1 Einleitung

Linda [1] ist eine Koordinationssprache, die ein flexibles Konzept zur Synchronisation und Kommunikation durch *Tupelräume* bietet. Als Erweiterung einer sequentiellen Programmiersprache (z.B. C) erhält man eine neue parallele Programmiersprache (z.B. C-Linda).

In der mengen-orientierten Prototyping-Sprache ProSet [2] wird das Konzept der Prozeßkreation durch Multilisp's Futures [3] auf die mengen-orientierte Programmierung zugeschnitten und mit dem flexiblen Konzept zur Koordination durch Linda's Tupelräume kombiniert. Diese Tupelräume sind virtuelle gemeinsame Datenräume, über die die Prozesse kommunizieren. ProSet-Linda wurde zum Prototyping paralleler Algorithmen entworfen [4]. Synchronisation und Kommunikation erfolgen in ProSet-Linda durch das Einfügen, Entfernen, Lesen und durch unteilbares Ändern von einzelnen Tupeln im Tupelraum. Der Zugriff erfolgt assoziativ und nicht über Adressen. Zum Finden eines Tupels wird ein sogenanntes *Template* angegeben, für das durch *Matching* ein passendes Tupel gesucht wird. In der vorliegenden Arbeit soll nun untersucht werden, welche Möglichkeiten und Konsequenzen sich durch eine Erweiterung des Matching hin zur Unifikation ergeben.

Zunächst zu den grundsätzlichen Unterschieden zwischen Unifikation und Matching. Zwei Terme s und t sind *unifizierbar*, falls es eine Substitution σ gibt, so daß $\sigma(s) = \sigma(t)$ gilt. $\sigma(s)$ wird dann als *Unifikation* von s und t bezeichnet. σ *unifiziert* s und t. Zwei Terme s und t *matchen*, falls es eine Substitution σ gibt, so daß $\sigma(s) = t$ gilt. Das *Matching* ist somit eine Variante der Unifikation. In Linda wird versucht, für ein Template *temp* ein Tupel *tup* zu finden, so daß $\sigma(temp) = tup$ gilt. Für einen Überblick über Anwendungen der Unifikation in verschiedenen Disziplinen sei auf [5] verwiesen.

In Abschnitt 2 wird Matching und ein Ansatz hin zur Unifikation in ProSet-Linda spezifiziert. In Abschnitt 3 diskutieren wir dann einige allgemeine Aspekte zur Unifikation in Tupelräumen.

2 Matching und Unifikation in ProSet-Linda

In ProSet-Linda kann ein Prozeß durch eine `deposit`-Operation ein Tupel in einen Tupelraum ablegen:

```
deposit [ "mystring", 123 ] at TS end deposit;
```

Diese Operation legt das Tupel `[ "mystring", 123 ]` in den Tupelraum `TS` ab. An dieser Stelle sei angemerkt, daß die Integration von Linda in ProSet sehr natürlich ist, da in beiden Sprachen Tupel eine wichtige Rolle spielen. Ein anderer Prozeß kann dann versuchen durch eine geeignete `fetch`-Operation dieses Tupel aus dem Tupelraum zu entfernen:

```
fetch ( "mystring", ? i ) at TS end fetch;
```

Das Laufzeitsystem versucht dann, für das Template `( "mystring", ? i )` durch Matching ein passendes Tupel zu finden. Die Template-Komponente `"mystring"` ist ein sogenanntes *Actual* (ein Wert). Die Template-Komponente `? i` ist ein sogenanntes *Formal*. Ein Tupel und ein Template *matchen*, falls die Anzahl der Komponenten gleich ist und falls die Actuals in den Templates gleich den entsprechenden Tupel-Komponenten sind. Die Variable `i` im obigen Formal erhält den Wert der entprechenden Tupel-Komponenten zugewiesen, nachdem ein passendes Tupel gefunden wurde. Diese Variable spielt für das Matching keine Rolle.

Wir verwenden hier die formale Spezifikationssprache Z [6] zur formalen Beschreibung von Matching und Unifikation in ProSet-Linda, um die bekannten Probleme mit informelle Spezifikationen — wie Mehrdeutigkeit und Unvollständigkeit — zu vermeiden. Für eine Einführung zu Z sei auf [6] verwiesen. Ein vollständige formale Spezifikation von ProSet-Linda ist in [7] mit Object-Z (eine objekt-orientierte Erweiterung von Z) angegeben. Wir geben hier nur einen vereinfachten Auszug aus dieser Spezifikation, soweit das zur Spezifikation von Matching und Unifikation in ProSet-Linda nötig ist. Zunächst definieren wir neue Basistypen für Werte und Formals:

$$[Value, Formal]$$

Auf diese Art werden neue Basistypen für eine Z-Spezifikation eingeführt, deren interne Struktur an dieser Stelle keine Rolle spielt und daher verborgen bleibt. Vordefiniert sind die Basistypen $\mathbb{N}$ und $\mathbb{Z}$. ProSet's undefinierter Wert `om` ist z.B. ein Wert:

$$om : Value$$

Für eine genaue Definition der Eigenschaften von Werten in ProSet sei auf [7] verwiesen.

Tupel in ProSet haben ihre übliche mathematische Bedeutung als geordnete Folgen von Werten. Ein Wert kann mehrfach in einem Tupel erscheinen. Die Reihenfolge ist relevant. Konzeptionell sind Tupel in ProSet unendliche Folgen von Werten, wobei fast alle Komponenten gleich dem undefinierten Wert `om` sind. Die Indizierung beginnt bei `1`. Die *Länge* eines Tupel, die durch ProSet's `#`-Operator geliefert wird, ist der größte Index einer Tupel-Komponente, die ungleich `om` ist (es gilt `#[1,om] = 1` und `#[om,1] = 2`). Da fast alle Komponenten gleich dem undefinierten Wert `om` sind, haben Tupel eine endliche Repräsentation und können mit Z als endliche Folgen von Werten definiert werden:

$$Tuple == \text{seq}\ Value$$

Die vordefinierte Funktion # aus Z kann hier nicht zur Spezifikation von ProSet's `#`-Operator (der ja die Länge eines Tupels liefert) verwendet werden, da `#[om] = 0` in ProSet gilt. Stattdessen definieren wir die Funktion *TupArity*, die die Länge eines Tupels spezifiziert:

$$TupArity : Tuple \rightarrow \mathbb{N}$$

$$\forall t : Tuple \bullet TupArity\ t = max(\{0\} \cup \{ i : \mathbb{N} \mid t(i) \neq om \})$$

Template-Komponenten sind Formals oder Actuals (Werte):

$$TempComp ::= TempValue\ \langle\langle Value \rangle\rangle \mid TempFormal\ \langle\langle Formal \rangle\rangle$$

Templates sind endliche Folgen von Template-Komponenten:

$$Template == \text{seq}\ TempComp$$

Die Funktion *TempArity* definiere dann analog zu *TupArity* die Länge von Templates. Als ersten Schritt definieren wir das Matching von individuellen Tupel- und Template-Komponenten:

$$\_\ \mathsf{CompMatches}\ \_ : Value \leftrightarrow TempComp$$

$$\textbf{let}\ ValueOfTemp == TempValue^{-1} \bullet$$
$$\forall tupc : Value;\ tempc : TempComp \bullet$$
$$tupc\ \mathsf{CompMatches}\ tempc \Leftrightarrow$$
$$(tempc \in \text{ran}\ TempValue \Rightarrow tupc = ValueOfTemp\ tempc)$$

Ein Tupel und ein Template *matchen* dann, falls die Anzahl der Komponenten gleich ist und falls die Actuals in den Templates gleich den entsprechenden Tupel-Komponenten sind:

$$
\begin{array}{|l}
\_ \ \mathsf{Matches} \ \_ : \mathit{Tuple} \quad \mathit{Template} \\
\hline
\forall\, \mathit{tup} : \mathit{Tuple};\ \mathit{temp} : \mathit{Template} \bullet \\
\quad \mathit{tup} \ \mathsf{Matches} \ \mathit{temp} \Leftrightarrow \\
\quad\quad \mathit{TupArity}(\mathit{tup}) = \mathit{TempArity}(\mathit{temp}) \land \\
\quad\quad (\forall\, i : 1 \,..\, \mathit{TupArity}(\mathit{tup}) \bullet \mathit{tup}(i) \ \mathsf{CompMatches} \ \mathit{temp}(i))
\end{array}
$$

CompMatches definiert, wann individuelle Tupel- und Template-Komponenten matchen. Matches definiert, wann ein Tupel und ein Template matchen.

Um das Matching nun in Richtung Unifikation zu erweitern, müssen als Tupel-Komponenten auch Formals zugelassen werden:

$$\mathit{TupComp} ::= \mathit{TupValue} \quad \mathit{Value} \quad | \quad \mathit{TupFormal} \quad \mathit{Formal}$$

Um die Unifikation im Tupelraum zu unterstützen, muß das Typsystem von ProSet also um Formals erweitert werden. Formals sind in ProSet bisher keine *first-class*-Objekte. Tupel sind dann endliche Folgen solcher Tupel-Komponenten:

$$\mathit{TupleU} == \mathrm{seq}\ \mathit{TupComp}$$

Die Funktion *TupArityU* definiere dann wieder analog zu *TupArity* die Länge solcher Tupel. Als ersten Schritt definieren wir wieder die Unifikation von individuellen Tupel- und Template-Komponenten:

$$
\begin{array}{|l}
\_ \ \mathsf{CompUnify} \ \_ : \mathit{TupComp} \quad \mathit{TempComp} \\
\hline
\textbf{let}\ \mathit{ValueOfTup} == \mathit{TupValue}^{-1}; \\
\quad \mathit{ValueOfTemp} == \mathit{TempValue}^{-1} \bullet \\
\forall\, \mathit{tupc} : \mathit{TupComp};\ \mathit{tempc} : \mathit{TempComp} \bullet \\
\quad \mathit{tupc} \ \mathsf{CompUnify} \ \mathit{tempc} \Leftrightarrow \\
\quad\quad (\mathit{tempc} \in \mathrm{ran}\ \mathit{TempValue} \land \mathit{tupc} \in \mathrm{ran}\ \mathit{TupValue} \Rightarrow \\
\quad\quad\quad \mathit{ValueOfTup}\ \mathit{tupc} = \mathit{ValueOfTemp}\ \mathit{tempc})
\end{array}
$$

Ein Tupel und ein Template *unifizieren* dann, falls die Anzahl der Komponenten gleich ist und falls die Actuals in den Templates gleich den Actuals in den entsprechenden Tupel-Komponenten sind (Formals passen immer):

$$
\begin{array}{|l}
\_ \ \mathsf{Unify} \ \_ : \mathit{TupleU} \quad \mathit{Template} \\
\hline
\forall\, \mathit{tup} : \mathit{TupleU};\ \mathit{temp} : \mathit{Template} \bullet \\
\quad \mathit{tup} \ \mathsf{Unify} \ \mathit{temp} \Leftrightarrow \\
\quad\quad \mathit{TupArityU}(\mathit{tup}) = \mathit{TempArity}(\mathit{temp}) \land \\
\quad\quad (\forall\, i : 1 \,..\, \mathit{TupArityU}(\mathit{tup}) \bullet \mathit{tup}(i) \ \mathsf{CompUnify} \ \mathit{temp}(i))
\end{array}
$$

CompUnify definiert, wann individuelle Tupel- und Template-Komponenten unifizieren. Unify definiert, wann ein Tupel und ein Template unifizieren. Durch Unify wird versucht, für ein Template *temp* und ein Tupel *tup* eine Substitution σ zu finden, so daß $\sigma(\mathit{temp}) = \sigma(\mathit{tup})$ gilt.

Die Erweiterung des Matching hin zur Unifikation ist also recht einfach möglich. Nachteilig scheint allerdings die Notwendigkeit zu sein, einen neuen Typ für Formals in die Sprache aufnehmen zu müssen. Die erhöhte Flexibilität für die Tupelraum-Kommunikation erhöht also auch die Komplexität der Sprache.

3 Abschlußbemerkungen

Die vorgestellte Erweiterung des Matching erhöht die Flexibilität von ProSet-Linda, indem sich für Sender von Tupeln einfache Möglichkeiten zum Broadcasting ergeben, ohne daß die Empfänger dazu

geändert werden müssen. Mit dem präsentierten Ansatz wurden natürlich noch nicht alle Möglichkeiten der Unifikation ausgeschöpft. Weitere Ergänzungen wären denkbar, um z.B. eine Markierung von Formals so zuzulassen, daß bestimmte Komponenten für eine erfolgreiche Substitution gleiche Werte haben müssen oder um Werte aus Actuals von Templates in Formals von Tupeln zu schreiben. Auch hier muß der Vorteil erhöhter Flexibilität gegenüber den Nachteilen erhöhter Komplexität abgewogen werden.

In Vorschlägen für Kombinationen von PROLOG mit Linda wird häufig Linda's Matching durch PROLOG's Unifikation ersetzt [8]. In gewisser Weise kann PROLOG's Datenbank von Regeln mit Linda's Tupelraum verglichen werden. Ein PROLOG-Ziel kann mit einem Template verglichen werden. Ein PROLOG-Interpreter versucht dann das Ziel mit der Datenbank von Regeln zu unifizieren, üblicherweise durch Backtracking. Ein Linda-System versucht Templates mit Tupeln zu matchen.

Ein wichtiger Unterschied zwischen Linda's Matching und PROLOG's Unifikation ist, daß Unifikation eine entsprechende Nachricht liefert, falls keine Substitution möglich ist, wohingegen Linda's Matching den entsprechenden Prozeß blockiert, falls kein passendes Tupel gefunden wurde. PROLOG's Voraussetzung einer "abgeschlossenen Welt" impliziert, daß eine Ausführung nicht auf fehlende Information warten kann. Ein weiterer Unterschied ist, daß PROLOG's Unifikation mehrere Substitutionen liefern kann. Durch Linda's Matching wird höchstens eine Substitution (ein Tupel) gefunden.

Weitere Probleme entstehen, falls in der Implementation Backtracking auch für Tupelraum-Operationen zugelassen wird. Backtracking von Kommunikations-Operationen unter parallelen Prozessen ist nur sehr schwierig zu implementieren. Auch die Spezifikation der Semantik wird dann problematisch. Für eine genaue Diskussion der resultierenden Probleme sei auf [9] verwiesen.

Literatur

[1] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Januar 1985.

[2] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers und C. Pahl. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos, Hrsg., *Proc. Third International Workshop on Rapid System Prototyping*, Seiten 235–248, Research Triangle Park, NC, Juni 1992. IEEE Computer Society Press.

[3] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oktober 1985.

[4] W. Hasselbring. Prototyping parallel algorithms with ProSet-Linda. In J. Volkert, Hrsg., *Parallel Computation (Proc. Second International ACPC Conference)*, Band 734 von *Lecture Notes in Computer Science*, Seiten 135–150, Gmunden, Austria, Oktober 1993. Springer-Verlag.

[5] K. Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, März 1989.

[6] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.

[7] W. Hasselbring. *Prototyping Parallel Algorithms in a Set-Oriented Language*. Dissertation, Universität Dortmund, 1994. (Erschienen im Verlag Dr. Kovač, Hamburg).

[8] K. De Bosschere, J.-M. Jacquet und P. Tarau, Hrsg. *Proc. ICLP'93 Post-Conference Workshop on Blackboard-Based Logic Languages*, Budapest, Hungary, Juni 1993.

[9] P. Ciancarini. Parallel programming with logic languages: a survey. *Computer Languages*, 17(4):213–240, April 1992.