7.5 Nebenläufige logische Programmierung: CCP

Logische Programmiersprachen haben ein hohes Potenzial für die parallele Implementierung:

- UND-Parallelismus: beweise alle Literale einer Anfrage parallel
- ODER-Parallelismus: benutze parallel alle Klauseln für ein Literal

Allerdings ist in diesen Fällen die Parallelität nicht sichtbar für den Programmierer, sondern nur ein Implementierungsaspekt. Dies ist anders bei *nebenläufigen logischen Programmiersprachen*:

- Diese enthalten explizite Konstrukte zur Nebenläufigkeit (z. B. Committed Choice, vgl. Kapitel 6.3.3).
- Sie wurden erstmalig entwickelt im Rahmen von Japans „Fifths Generation Computing Project".
- Entsprechende Berechnungsmodelle wurden in vielen verschiedenen Sprachen realisiert.
- Saraswat ((Saraswat, 1993)) hat dieses weiterentwickelt und verallgemeinert zum CCP-Modell, welches wir im folgende vorstellen.

CCP steht hierbei für **Concurrent Constraint Programming**:

- Es ist eine Erweiterung von CLP um Nebenläufigkeit.
- Die Synchronisation erfolgt durch Gültigkeit von Constraints.
- Das Berechnungsprinzip sind nebenläufige Agenten (≈ Prozesse), die auf einem gemeinsamen **Constraint-Speicher** (CS) arbeiten.
- Jeder Agent kann dabei im Wesentlichen folgendes machen:
  - Er wartet auf Information in CS („**ask**"), oder
  - er fügt neue (konsistente) Information zu CS hinzu („**tell**")

  Somit entspricht dies von der Idee dem Linda-Modell ohne „`in`", d.h. Löschen von Informationen, allerdings werden statt Tupel Constraints verwendet, wodurch die Berechnungen wie in der Logikprogrammierung auf prädikatenlogischen Grundlagen beruht.
- Der Constraint-Speicher CS wird nur mit Information angereichert, aber es wird nichts entfernt. Dies hat die folgende wichtige Konsequenzen:
  - Die Reihenfolge der Agentenbearbeitung kann andere Agenten nicht blockieren.
  - Hierdurch werden Deadlocks, die nur auf Grund eines ungünstigen Scheduling entstehen, automatisch vermieden.

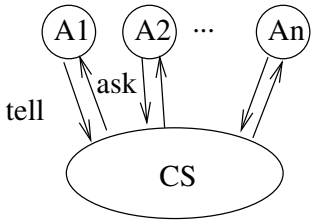

Abbildung 7.4: Skizze von CCP mit Agenten und Constraint-Speicher

Beispiel 7.14 (Maximum-Agent)**.**

```
max(X,Y,Z) := X <= Y | Y=Z
            □ X >= Y | X=Z
```

Bedeutung der synaktischen Konstrukte:

- links von „|“: „ask“-Constraints (entspricht „`rd`“ in Linda)
- rechts von „|“: „tell“-Constraints (entspricht „`out`“ in Linda)
- □ kennzeichnet Alternativen (committed choice)

Der Agent `max(X,Y,Z)`

- wartet, bis Relation zwischen `X` und `Y` aus CS bekannt, und
- wählt dann **eine** passende Alternative und fügt ein Gleichheitsconstraint zu CS hinzu.

Damit können die folgenden möglichen Situationen entstehen („$\models$“ bezeichnet die logische Implikation aus dem Constraint-Speicher):

- `CS` $\models$ `X=1` und `CS` $\models$ `Y=3`: Füge `Z=3` zu CS hinzu
- `CS` $\models$ `X=Y+2`: Füge `X=Z` zu CS hinzu

Beispiel 7.15 (Nichtdeterministischer Strommischer)**.** Aufgabe: Mische Ströme (Listen), sobald Teile bekannt sind.

```
merge(S1,S2,S) := S1=[]      | S=S2
                □ S2=[]      | S=S1
                □ S1=[E|ST1] | S=[E|ST], merge(ST1,S2,ST)
                □ S2=[E|ST2] | S=[E|ST], merge(S1,ST2,ST)
```

Mit diesem Mischer kann man Kommunikationsstrukturen zwischen mehreren Agenten, die `ask`/`tell` verwenden, aufbauen. Daher ist CCP auch geeignet für die Implementierung von Multiagentensystemen. Eine solche Struktur wollen wir kurz skizzieren:

Beispiel 7.16 (Client/Server-Programmierung)**.** Der Client/Erzeuger hat folgende Programmstruktur:

```
client(S,...) := ... | compute(E), S=[E|S1], client(S1,...)
```

Der Server/Verbraucher hat diese Programmstruktur:

```
server(S,...) := S=[E|S1] | process(E), ..., server(S1,...)
```

Dann können wir einen Server mit zwei Clients wie folgt bauen:

```
client(S1,...), client(S2,...), merge(S1,S2,S), server(S,...)
```

Eine wichtige Idee von CCP ist die Unterstützung eines *deklarativen Konzepts von Nebenläufigkeit*: zwar sind hier auch Deadlocks prinzipiell möglich, allerdings können keine Deadlocks wegen eines ungünstigen Schedulings entstehen, die bei einem anderen Scheduling nicht auftreten. Dies liegt insbesondere daran, dass zum Constraint-Speicher nur Information hinzugefügt wird, aber keine Information gelöscht wird.