Diplomarbeit

Referenznetze mit Anschriften in Scheme

Friedrich Delgado Friedrichs

27. September 2007

Universität Hamburg, Department Informatik
Betreuer:
Dr. Michael Köhler
Prof. Dr. Leonie Dreschler-Fischer

Inhaltsverzeichnis

Einleitung

Referenznetze sind objektbasierte, gefärbte Petrinetze, die Referenzen auf Netze als Marken enthalten und über synchrone Kanäle kommunizieren können.

Mit der Petrinetz-Entwicklungsumgebung RENEW[1] können komplexe Anwendungen mit Referenznetzen implementiert werden.

Wenn Referenznetze mit Java-Beschriftungen (im Folgenden „Java-Referenznetze“ genannt) mit RENEW simuliert werden, können sie beim Schalten von Transitionen beliebige Java-Programme ausführen, wodurch die mächtigen Synchronisationsmechanismen der Petrinetze für den Java-Programmierer zur Verfügung stehen, bzw. alle in Java lösbaren Aufgaben auch mit Java-Referenznetzen umsetzbar werden.

Die Programmiersprache Scheme unterstützt neben vielen anderen Paradigmen auch die funktionale Programmierung. Die Sprache erlaubt es, den Umfang der zur Verfügung stehenden Bindungen auf eine Untermenge einzuschränken. Sie kann aber auch so erweitert werden, dass z. B. deklarative Programmierung oder die Verwendung von Java-Objekten möglich wird. Des Weiteren ist sie eine der wenigen Programmiersprachen, deren Standard eine formale Semantik des Sprachkerns festlegt, was eine formale Behandlung von Referenznetzen mit Scheme-Beschriftungen (im Folgenden „Scheme-Referenznetze“ genannt) prinzipiell erlauben sollte.

Diese Arbeit beschreibt die Syntax und das Verhalten von Referenznetzen mit Anschriften in der Programmiersprache Scheme sowie ein neues Plugin für RENEW, mit dem Netzanschriften für Refe-

[1] http://renew.de/ Alle URLs in dieser Arbeit wurden zuletzt im September 2007 besucht.

renznetze in der Programmiersprache Scheme erstellt und die damit beschriebenen Scheme-Referenznetze simuliert werden können.

Mit diesem Scheme-Plugin können bereits komplexere Beispiele für Java-Referenznetze in den Scheme-Formalismus übertragen werden und verhalten sich im RENEW-Simulator wie ihre Java-Vorbilder. Mit wenigen noch fehlenden Erweiterungen können Scheme-Referenznetze also die gleiche Ausdrucksmächtigkeit wie Java-Referenznetze erreichen.

Darüber hinaus ergeben sich durch die Sprache Scheme als Beschriftungssprache weitere Ausdrucksmöglichkeiten, von denen einige ebenfalls in der Arbeit angedeutet werden.

Als mögliche Ergänzung zum RENEW-Plugin wird das Konzept einer Einbettung von Scheme-Referenznetzen in die Programmiersprache Scheme in Grundzügen skizziert.

Typografische Konventionen

Quellcode, URLs, Kommandos sowie Prozessnamen werden in dieser Arbeit in `teletype` Schrifttypen gesetzt, die Namen von Programmpaketen, der LaTeX Konvention folgend, in KAPITÄLCHEN (und Akronyme demzufolge in Großbuchstaben). Aus diesem Grund findet sich in diesem Text z. B. die Schreibweise RENEW, wenn das Programmpaket gemeint ist und `renew` wenn das konkrete Kommando oder der Prozess gemeint ist.

Danksagungen

Ich danke meinem Arbeitgeber, der DFN-CERT Services GmbH, Hamburg, für die ausgesprochen flexible Gestaltung meiner Arbeitszeiten und die Unterstützung beim Beenden meines Studiums. Ohne diese Unterstützung wäre diese Arbeit nicht zu Stande gekommen. Ich danke Taylor R. Campbell für seine Hilfe mit den Feinheiten der Sprache Scheme und seiner Makros, Scott G. Miller und Matthias Radestock für ihre Hilfestellung und dafür, dass sie genau das

richtige Werkzeug für meine Idee erschaffen haben.[2] Ich danke meinen Korrekturlesern Axel Großklaus, Susanne Schmitz und Jochen Schönfelder. Alle noch in dieser Arbeit verbliebenen Fehler liegen allein in meiner Verantwortung.

Vorgehen

Die Arbeit gliedert sich in vier Hauptteile.

Zunächst werden die Ausgangssituation und die sich daraus ergebenden Motivationen und Zielsetzungen im Teil „Ausgangssituation“ (S.3) beschrieben.

Die Kapitel im Teil „Konzeption“ (S.29) behandeln die Anforderungen an den Scheme-Formalismus für Referenznetze, sowie Vorgaben für ihre Umsetzung im Hinblick auf mögliche Implementationen aber noch *möglichst* unabhängig von einer konkreten Implementation.

Der nächste Teil „Umsetzung“ (S.123) greift diese Vorgaben auf und beschreibt eine im Rahmen dieser Arbeit erstellte konkrete Implementation des Formalismus für die Petrinetz-Entwicklungsumgebung RENEW.

Im vierten und letzten Teil „Ergebnis“ (S.177) wird das Ergebnis vorgestellt und bewertet. Eine Anleitung zur Verwendung der erstellten Implementation gliedert sich ebenfalls im letzten Teil ein.

[2] “I would like to thank Taylor R. Campbell for his help with the finer details of the programming language scheme and its macros, Scott G. Miller and Matthias Radestock for their assistance, and for creating exactly the right tool for realising my idea.” I’ll prepare an announcement for the scheme community and some english documentation a bit later.

Teil I

Ausgangssituation

1 Java-Referenznetze in RENEW

Dieses Kapitel stellt Java-Referenznetze vor und demonstriert anhand von Beispielen, dass Nebeneffekte in Referenznetzen das Lokalitätsprinzip für Petrinetze verletzen. Im Konzept der Scheme-Referenznetze soll darum ein Mechanismus zur Beschränkung von Nebeneffekten mit eingeplant werden.

Teile dieses Kapitels wurden meiner Studienarbeit [Delgado Friedrichs, 2006] entnommen.

1.1 RENEW, Referenznetze und Java

Zunächst gebe ich die informelle Definition der Referenznetze aus der vorgenannten Arbeit wieder, vornehmlich im Hinblick auf die Java-Beschriftungen.

RENEW

Definition 1. RENEW *ist ein Werkzeug zur Entwicklung und Ausführung objektorientierter Petrinetze mit Netzinstanzen, synchronen Kanälen und nahtloser Integration der Programmiersprache Java zur vereinfachten Modellierung (nach [Kummer u. a., 2004]). Es wurde am Fachbereich Informatik der Universität Hamburg von Olaf Kummer, Frank Wienberg, Michael Duvigneau und einigen anderen entwickelt. Das Programm ist mit dem kompletten Java-Quellcode als Open-Source Software verfügbar und kann unter* `http://www.renew.de` *bezogen werden.*

Referenznetze

Definition 2. ***Referenznetze****, wie sie in* RENEW *implementiert werden, sind gefärbte Petrinetze, die über synchrone Kanäle verfügen und deren Markierungen neben anderen (Java-)Datenobjekten auch Referenzen auf Netzexemplare enthalten können.*

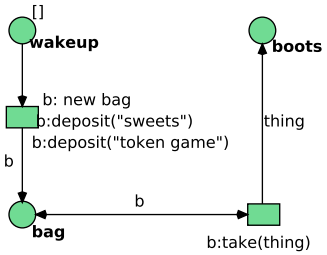

Abbildung 1.1: Das Netz „santa“

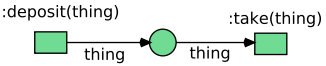

Abbildung 1.2: Das Netz „bag“

Durch synchrone Kanäle können zum Einen Transitionen in Referenznetzen miteinander synchronisiert werden und zum Anderen wird (über die Kanalparameter) Datenübertragung zwischen Netzexemplaren ermöglicht.

In den Beispielnetzen[1] „santa“ und „bag“ (Abbildungen 1.1 und 1.2) werden diese beiden Mechanismen illustriert. In der Transition zwischen den Stellen „wakeup“ und „bag“ im Netz „santa“ wird zunächst eine Referenz auf ein neues Netzexemplar des Netzmusters „bag“ erzeugt[2] und dann zwei mal der Downlink `bag:deposit()` aufgerufen[3]. Dadurch schaltet die Transition mit dem Uplink `:deposit(thing)` in der erzeugten Netzinstanz `b` und das dem Kanalparameter übergebene Datenobjekt wird an den formalen Parameter `thing` gebunden und in der Ausgangsstelle der Transition im Netzexemplar `b` hinterlegt. Der Bindungsmechanismus ist hierbei eine Form des Pattern-Matching bzw. der Unifikation, der mit Hilfe eines Unifikationsalgorithmus realisiert wird[4].

[1] aus [Kummer u. a., 2006b]

[2] Terminologie nach [Kummer, 2002]. Netzexemplare sind konkrete Instanzen eines einzigen abstrakten Netzmusters, das u. U. in mehrfachen Exemplaren instantiiert sein kann.

[3] Die Reihenfolge in der die Transitionsanschriften evaluiert werden ergibt sich nebenbei bemerkt **nicht** aus der textuellen Reihenfolge. Siehe dazu [Kummer u. a., 2006b] und für eine detaillierte Beschreibung der Algorithmen [Kummer, 2002].

[4] Es wird in dieser Arbeit oft etwas unpräzise von Unifikation geredet, wo Pat-

Neben der Erzeugung und Verwendung von Netzreferenzen als Marken und der Kommunikation über synchrone Kanäle können in RENEW viele Arten von Kanten und Anschriften verwendet werden. Reservekanten, Testkanten und flexible Kanten sowie virtuelle Stellen sind bereits aus anderen Petrinetzformalismen bekannt.

1.2 Java-Anschriften für Referenznetze

Die Beschriftungssprache für Java-Referenznetze unterscheidet sich vom Standarddialekt in einigen Punkten, die hier nach [Kummer u. a., 2006b] zusammengefasst werden.

Der Großteil der Abweichungen wurde eingeführt, um Nebeneffekte zu vermeiden, insbesondere dort wo sie mit der Semantik von Petrinetzen kollidieren[Kummer, 2002, Kapitel 12]. Im Allgemeinen sollte die Möglichkeit *eingeschränkt* werden, dass Anschriften eine bestimmte Auswertungsreihenfolge oder Zustandsänderungen außerhalb der Lokalität der schaltenden Transition zur Folge haben können. Es ist aber unmöglich, solche Nebeneffekte in Java Anschriften vollständig zu vermeiden, weswegen im RENEW-Handbuch [Kummer u. a., 2006b] mehrfach vor möglichen Problemen gewarnt wird.

Eine Besonderheit Java-basierter Referenznetze sind die „Action-Anschriften“ mit denen beim Schalten einer Transition *beliebige* Java-Ausdrücke ausgeführt werden können. (Siehe wiederum [Kummer u. a., 2006b] für die praktische Verwendung und [Kummer, 2002] für die theoretische Fundierung.) Das heißt in den Action-Anschriften sind einige der unten angegebenen Beschränkungen aufgehoben und dem Benutzer wird nahe gelegt, die Erzeugung von

tern-Matching ausreichend ist. Das hängt zum Einen damit zusammen, dass auch zum Pattern-Matching eine Bibliothek verwendet werden soll, mit der Unifikation möglich ist, zum Anderen aber auch damit, dass sich bei Scheme-Referenznetzen nur im Spezialfall entscheiden lässt, ob Pattern-Matching oder Unifikation nötig ist, insbesondere da ein spezieller Unifikationsoperator eingeführt wird, der tatsächlich Unifikation an Transitionen durchführt.

Nebeneffekten soweit möglich auf Action-Anschriften zu beschränken.

Andere Abweichungen vom Java Standard [Gosling u. a., 2005] sind die folgenden:

- Die Syntax basiert auf der von Java 1.0.2. Das bedeutet z. B. dass innere Klassen (Java 1.2) oder Templates (Java 1.5) nicht verwendet werden können.

- Kurzschluss-Logik Operatoren (d.h. `&&`, `||` und `?:`) wurden entfernt, weil sie eine Ausführungsreihenfolge vorgeben, deren Semantik bei der Unifizierung nicht eindeutig bestimmbar ist.

- Außerhalb von Action-Anschriften ist der Operator = eine Gleichheitsspezifikation an das Unifikationsverfahren. Die zu unifizierenden Objekte werden mit der Java-Methode `equals` verglichen. Eine Auswertungsreihenfolge wird nicht vorgegeben und die Variablen an Kanten- oder Transitionsanschriften können für jeden Modus der Transition an unterschiedliche Werte gebunden werden. Eine Besonderheit dieses Operators ist weiterhin, dass mit seiner Hilfe in Java-Referenznetzen *neue* Bezeichner eingeführt werden können. (Diese Eigenschaft wird in den Arbeiten zu Referenznetzen und RENEW i. d. R. nicht besonders hervorgehoben, ergibt sich aber aus der Sonderstellung der Gleichheitsspezifikation = als direkte Anweisung an den Unifikationsalgorithmus. Weitere Diskussion siehe Abschnitt 7.5.1 (S. 74).)

- Bezeichner in Java-Referenznetzen können ungetypt sein.

- Die Notation `[`x_1`,` x_2`, ...]` bezeichnet ein Tupel mit den Elementen x_1, x_2, usw. Das leere Tupel `[]` wird in RENEW als „Black Token“ (Marke ohne Eigenschaften) verwendet. Tupel sind polymorph, d.h. die Tupelelemente müssen nicht den gleichen Typ haben. Wenn, z. B. in Guard-Ausdrücken, auf

einzelne Elemente von zu unifizierenden Tupeln Bezug genommen wird, müssen diese die gleiche Anzahl Elemente aufweisen.

- Die Notation `{`x_1`, `x_2`, ...}` bezeichnet eine Liste mit den Elementen x_1, x_2, usw. Listen sind ebenfalls polymorph. Darüber hinaus können Listen ohne Berücksichtigung ihrer Länge miteinander unifiziert werden. Dazu steht der Operator `:` zur Verfügung, der eine Liste aus dem Kopf und dem Rest der Liste konstruiert.

- Des Weiteren wird eine Notation für die bereits erwähnten Netzexemplare und synchronen Kanäle bereitgestellt. Da neue Netzexemplare nur an Transitionen erzeugt werden können und dort direkt an Bezeichner gebunden werden *müssen*, wurde von der üblichen Java-Notation `new Konstruktor(...)` abgewichen. Die Notation für die Erzeugung eines neuen Netzexemplars wurde mit der Notation für Downlinks vereinheitlicht. Siehe Abbildung 1.1 (S. 4).

- In den Deklarationsknoten von Java-Referenznetzen sind nur Import- und Variablendeklarationen (ohne Initialisierung) erlaubt. Methoden und Klassen, die in Java-Referenznetzen verwendet werden sollen, müssen also außerhalb der Netze spezifiziert werden[5].

1.3 Nebeneffekte in Referenznetzen

Bereits in der Einleitung habe ich angedeutet, dass Nebeneffekte in Java-Referenznetzen für die Analyse und Verifikation proble-

[5] Auch der Versuch, als initiale Markierung einer Stelle eine anonyme Implementierung eines Java-Interfaces zu erzeugen, wird mit einem Syntaxfehler „no such Constructor …“ quittiert, da innere Klassen in der auf Java 1.0.2 basierenden Syntax nicht bekannt sind.

matisch sein können. Die folgenden Unterabschnitte erläutern verschiedene Möglichkeiten wie Nebeneffekte erzeugt werden können, insbesondere solche, die dem Prinzip der Lokalität in Petrinetzen widersprechen.

Was das Prinzip der Lokalität speziell in Referenznetzen genau bedeutet, wird anschließend noch genauer untersucht.

Nebeneffekt

Ein **Nebeneffekt** tritt auf, wenn in der Umgebung eines Programms ein Zustand verändert wird.

In Bezug auf Referenznetze sind hier zunächst nur Veränderungen von sichtbaren Bindungen relevant. Problematisch sind bei Referenznetzen solche Zustandsänderungen, die in einer Teilstruktur einer Marke auftreten (wie etwa eine Instanzenvariable eines Objekts), ohne dass dabei die Markierung an sich verändert wird (d. h. es befindet sich weiterhin die gleiche Referenz in der Stelle).

Die Problematik ergibt sich allgemein daraus, dass die Anschriftsprache es erlaubt den Zustand eines Objekts zu verändern und diese Veränderungen wiederum in anderen Anschriften sichtbar sind und in Guards getestet werden können, so dass sie Auswirkungen auf die Aktivierung einer Transition haben.

In allen Beispielnetzen dieses Abschnitts wird durch zwei unterschiedliche Transitionen ein gemeinsamer Zustand verändert, was von der Netzsemantik her zwei mögliche Endzustände des jeweiligen Netzes zur Folge hat, je nachdem welche Transition zuerst schaltet. Welche Transition zuerst schaltet ist nicht deterministisch. Dies führt zum Einen dazu, dass jeweils nur in einem der beiden „Zweige“ des Netzes die jeweils letzte Stelle markiert wird. Dieses Verhalten ist nach der Spezifikation der (Java-)Referenznetze korrekt.

Zum Anderen führt eine Eigenheit des RENEW-Simulators dazu, dass in den in Abbildungen 1.3 auf der folgenden Seite und 1.4 (S. 10) gezeigten Netzen unter Umständen *keiner* der beiden Endzustände erreicht wird, was semantisch inkorrekt ist. Auf diese Problematik wird aber im RENEW-Handbuch hingewiesen [Kummer u. a., 2006b, S. 57]. Weitere problematische Fälle wurden durch die Konstruktion der Netze ausgeschlossen, die sich, abgesehen von dem

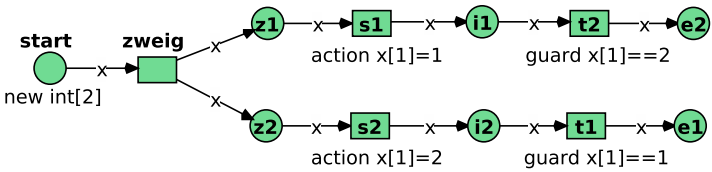

Abbildung 1.3: Ein Beispielnetz für Nebeneffekte von Transitionen mit „`action`“-Anschriften

gerade angesprochenen Sonderfall, im Simulator identisch verhalten sollten. (Wenn alle Transitionen von Hand ausgelöst werden, tritt der Sonderfall nicht auf.)

Das erwartungsgemäße Verhalten ist nicht unbedingt problematisch. In Abschnitt 1.3.4 (S. 12) gehe ich darauf ein, wie das Verhalten der Netze in Bezug auf eine mögliche Definition des Lokalitätsprinzips für Referenznetze zu bewerten ist. Im Kapitel 4 werden mögliche Lösungsansätze für die beschriebenen Probleme gesucht.

1.3.1 Nebeneffekte mit `action`-Anschriften

In Abbildung 1.3 wird in der Stelle **`start`** ein Java-Array vom Typ *`int`* mit zwei Einträgen angelegt.[6] Eine Referenz auf dieses Feld wird nach dem Schalten der Transition **`zweig`** in den linken Stellen der beiden „Gabelungsarme“ des Netzes hinterlegt (**`z1`** und **`z1`**). Wäre dieses Netz ein S/T-Netz würde man auf Grund der grafischen Darstellung ohne Zögern annehmen, dass alle Transitionen in den beiden Zweigen zueinander nebenläufig schalten können.

Allerdings verändern die beiden Transitionen **`s1`** und **`s2`** den Zustand des Arrays in widersprüchlicher Weise. Je nachdem ob **`s1`** oder **`s2`** zuerst schaltet hat das Feld am Index 1 den Wert 1 oder

[6] Ein Array oder ein anderer Datentyp mit innerer Struktur muss in diesem Beispiel verwendet werden, damit die Referenz auf `x` an den Transitionen konstant bleibt. Bei einer Zuweisung wie etwa `action x=2` würde RENEW ein `Calculator`-Objekt für `x` anlegen, das in diesem Fall nicht mehr mit dem Wert in der Stelle unifizierbar wäre.

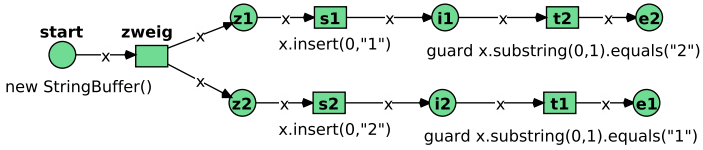

Abbildung 1.4: Ein Beispielnetz für Nebeneffekte von Transitionen mit Methodenaufrufen

2 und das wirkt sich auf die Aktivierung der Transitionen **t2** und **t1** aus. Die letzte Transition, die schalten kann ist also entweder **t2** oder **t1** und demgemäß enthält am Ende entweder die Stelle **e2** oder die Stelle **e1** eine Referenz auf das Feld.

Theoretisch wäre es sogar denkbar, dass **s1** und **s2** gleichzeitig schalten, das Netz wurde aber so konstruiert, dass dies keine Auswirkungen hat.

Soweit jedenfalls das erwartete Verhalten.

Der RENEW-Simulator prüft veränderliche Eigenschaften von Java-Objekten aber nur, wenn sich entweder die Markierung der betreffenden Stelle ändert, oder der Benutzer eine erneute Prüfung anfordert.

Das führt gelegentlich dazu, dass keine der Transitionen **t1** oder **t2** mehr schalten kann. Dieser Effekt tritt nur auf, wenn das Netz mit [„`Run Simulation (Strg-R)`“] gestartet oder in Einzelschritten simuliert wird [„`Simulation Step (Strg-I)`“]. Dann lässt er sich aber sehr häufig beobachten, weil er *stets* dann auftritt, wenn nach dem Schalten von **s2** geprüft wird ob **t1** schalten kann, bzw. wenn nach dem Schalten von **s1** geprüft wird, ob **t2** schalten kann.

Diese Eigenschaft des Simulators ist für die Betrachtungen der Nebeneffekte in Referenznetzen *nicht* relevant. Sie wird hier nur erwähnt, damit das Verhalten der Netze an dieser Stelle vollständig und richtig beschrieben wird.

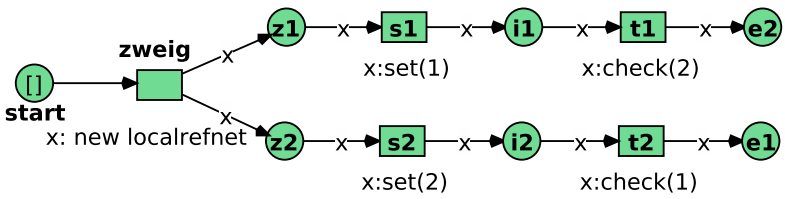

Abbildung 1.5: Ein Beispiel für Nebeneffekte mit synchronen Kanälen

1.3.2 Nebeneffekte mit Methodenaufrufen

Das Netz in Abbildung 1.4 auf der vorhergehenden Seite verhält sich fast identisch zum vorher besprochenen Netz, jedoch wird hier der Nebeneffekt mit Hilfe eines Methodenaufrufs erzeugt. Methoden können auch ohne das Schlüsselwort `action` aufgerufen werden, was sogar (mit Einschränkungen) in Kantenanschriften möglich ist. Im Handbuch [Kummer u. a., 2006b] wird ausdrücklich davon abgeraten.

Es gibt aber einen subtilen semantischen Unterschied zwischen den Netzen in den Abbildungen 1.3 und 1.4. Die Array-Zuweisung in Java ist nicht gegen nebenläufige Ausführung gesichert, die Methoden `insert` und `substring` der `StringBuffer` Klasse hingegen sind synchronisiert (*`synchronized`*), also tritt in diesem Netz das zusätzliche Problem auf, dass die Transitionen **`s1`**, **`s2`**, **`t1`** und **`t2`** bedingt durch die Semantik der verwendeten Methoden nicht nebenläufig zueinander schalten können, was aber für den RENEW-Simulator nicht erkennbar ist.

Dennoch ist in diesem Fall das Verhalten im Simulator identisch, hier kommt es neben den beiden Markierungen in **`e1`** und **`e2`** auch dazu, dass **`i1`** und **`i2`** markiert sind aber **`t1`** und **`t2`** niemals schalten.

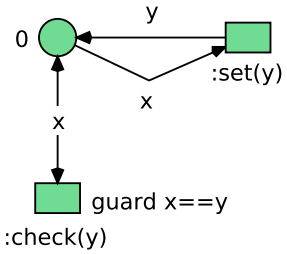

Abbildung 1.6: Das Netz „localrefnet“

1.3.3 Nebeneffekte mit Netzexemplaren und synchronen Kanälen

Im Netz in Abbildung 1.5 auf der vorigen Seite können ebenfalls die Transitionen **`s1`** und **`s2`** nicht nebenläufig schalten, in diesem Fall ist dies aber für den Simulator erkennbar. Statt der Reservekante hätte ich im Netz `localrefnet` in Abbildung 1.6 eine Testkante verwenden können, aber die beiden Transitionen **`t1`** und **`t2`** können ohnehin durch den Guard-Ausdruck in allen Beispielnetzen nicht nebenläufig schalten, das Verhalten der Transitionen verändert sich also tatsächlich nicht.

Im Gegensatz zu den beiden vorhergehenden Netzen ist das Verhalten des Referenznetzes in Abbildung 1.5 im Simulator nicht mehr problematisch. Im Simulator wird in jedem Fall schließlich eine der beiden Stellen **`e1`** oder **`e2`** markiert, da die Überprüfung von synchronen Kanälen durch den Simulator vorgesehen ist.

1.3.4 Wann und warum sind Nebeneffekte in Referenznetzen problematisch?

Das letzte Beispiel leitet die Diskussion ein.

Das Netz in Abbildung 1.5 auf der vorigen Seite verhält sich identisch zu dem in Abbildung 1.7 auf der folgenden Seite, wenn man von der zusätzlichen Stelle absieht, die statt der veränderlichen Ob-

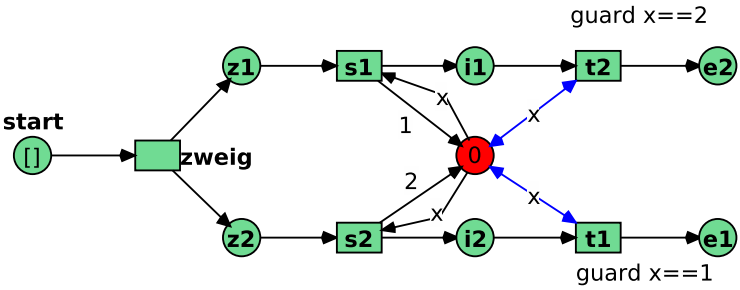

Abbildung 1.7: Ein Beispielnetz für Nebeneffekte von Transitionen unter Einhaltung des Lokalitätsprinzips für gefärbte Netze

jekte eingeführt wurde. Die Markierung von **`e1`** oder **`e2`** wird in jedem Fall erreicht. Die zusätzlichen Kanten und die Stelle mit der Anfangsmarkierung „`0`“ in der Mitte bilden die Zustandsänderung nach. Außerdem ist in diesem Netz die Lokalität aller Transitionen klar erkennbar und man erkennt, dass die beiden „Zweige“ in allen vorangegangenen Netzen eben nicht nebenläufig zueinander schalten können.

Der einzige „Nebeneffekt“ in diesem Netz besteht darin, dass die Markierung der Stelle mit der Anfangsmarkierung `0` verändert wird, was sowohl bei gefärbten Netzen als auch bei Referenznetzen ausdrücklich ein erwünschter Effekt ist.

Die Netze in Abbildung 1.5 und 1.7 unterscheiden sich aber in einem wichtigen Punkt: Für das letztere gibt es eine Anpassung des Lokalitätsbegriffs und das Lokalitätsprinzip wird eingehalten, für das erstere gilt dies nicht.

Was heißt Lokalität in Referenznetzen?

In [Kummer, 2002] wird keine explizite Anpassung des Lokalitätsbegriffs für Referenznetze vorgenommen.

Das Prinzip der Lokalität für Petrinetze wird in Valk [2003a] fol-

gendermaßen definiert[7]:

Prinzip der Lokalität für Petrinetze

Definition 3. ***Prinzip der Lokalität für Petrinetze****: Das Verhalten einer Transition wird ausschließlich durch ihre Lokalität bestimmt, welche sich aus ihr und der Gesamtheit ihrer Eingangs- und Ausgangs-Elemente (Vor- und Nachbedingungen, Ein- und Ausgabestellen) zusammensetzt.*

In [Valk, 2003a, S.15] findet sich eine formale Definition der Lokalität einer Transition durch „die Vereinigung von Vor- und Nachbereich und der Transition selbst“ ($loc(t) := \{t\} \cup {}^\bullet t \cup t^\bullet$).

Nebenläufigkeitsprinzip für Petrinetze

Nebenläufigkeitsprinzip für Petrinetze[8]: Transitionen mit disjunkter Lokalität ($loc(t_1) \cap loc(t_2) = \emptyset$) können unabhängig (nebenläufig) schalten (ebd.).

Später wird festgestellt, dass das Lokalitätsprinzip für gefärbte Netze erhalten bleibt und dort die Lokalität einer Transition „als alle Stellen, Kanten und Guards, die mit dieser Transition verbunden sind“ verstanden werden sollte [Valk, 2003b, S.38][9]. Das heißt, das Verständnis des Lokalitätsbegriffs wird für gefärbte Netze um die (hinzu getretenen) Guards erweitert.

Der Lokalitätsbegriff hat also den Nutzen, dass sich mit seiner Hilfe leichter Entscheidungen über das Verhalten von Netzen treffen lassen. Dies wird unter anderem auch bei der automatischen Analyse ausgenutzt (siehe z. B. [Kristensen und Christensen, 2004]; bei den CPN TOOLS können aber unterschiedliche Bindungen der gleichen Transition nebenläufig aktiviert sein und auch nebenläufig schalten).

[7] Im Originaltext: **The Principle of Locality for Petri Nets**: The behaviour of a transition exclusively depends on its *locality*, which is defined as the totality of its input and output objects (pre- and post-conditions, input and output places, …) together with the Element itself.

[8] Originaltext: **The Principle of Concurrency for Petri Nets**: Transitions having disjoint locality occur independently (concurrently)

[9] Originaltext: […] (which should be understood here as all places, arcs, and guards connected with this transition) […]

Es ist wichtig festzustellen, dass die Farben bei gefärbten Petrinetzen durch Wertemengen im mathematischen Sinne definiert sind. Im Unterschied zu Objekten einer imperativen Programmiersprache haben mathematische Werte in aller Regel keinen veränderlichen Zustand.

Das Netz in Abbildung 1.7 (S. 13) lässt sich als gefärbtes Netz unter Verwendung der natürlichen Zahlen mit einem Operator == als Farbmenge interpretieren. Damit haben die Lokalitäten der Transitionen **`s1`**, **`s2`**, **`t1`** und **`t2`** eine nicht-leere Schnittmenge und können *nicht* nebenläufig zueinander schalten.

Das Verhalten des Netzes in Abbildung 1.5 (S. 11) ist in jeder Hinsicht identisch. Das legt die folgende Erweiterung des Lokalitätsbegriffs für Referenznetze nahe, die der bereits in [Delgado Friedrichs, 2006] vorgeschlagenen ähnelt:

Definition 4. *Die **Lokalität einer Transition in einem Referenznetzexemplar** besteht*

Lokalität einer Transition in einem Referenznetzexemplar

1. *aus allen Stellen, Kanten und Guards, die mit dieser Transition verbunden sind,*
2. *sowie aus allen synchronen Kanälen, die mit dieser Transition verbunden sind,*
3. *vereinigt mit den Lokalitäten aller an diesen synchronen Kanälen beteiligten Transitionen.*

Oben wird anhand des Lokalitätsbegriffs für gefärbte Netze festgestellt, welche Transitionen im Netz in Abbildung 1.7 nebenläufig zueinander schalten können.

Mit Hilfe des vorgeschlagenen Lokalitätsbegriffs für Referenznetze lassen sich nun entsprechende Aussagen über das Netz in Abbildung 1.5 treffen: Die Transitionen **`s1`** und **`s2`** sind über den Kanal `set` und die Transitionen **`t1`** und **`t2`** sind über den Kanal `check` mit der Stelle im Netz `localrefnet` verbunden, also haben ihre Lokalitäten ein gemeinsames Element. Sie können also *nicht* nebenläufig zueinander schalten. Dies entspricht genau dem Ergebnis für das Netz in Abbildung 1.7.

Man kann mit dieser Definition also feststellen, dass das Lokalitätsprinzip im Netz in Abbildung 1.5 (S. 11) eingehalten wird, während es bei den ihm vorangehenden Beispielnetzen dieses Unterabschnitts verletzt wird.

Die Lokalität einer Transition lässt sich damit an der grafischen oder formalen Darstellung des Netzes, unter Berücksichtigung der synchronen Kanäle und Guards explizit ablesen.

Ist der vorgeschlagene Lokalitätsbegriff für Referenznetze ausreichend?

Definition 4 auf der vorhergehenden Seite berücksichtigt im Gegensatz zu dem Vorschlag in [Delgado Friedrichs, 2006] keine eventuell durch die Anschriftsprache erzeugten Nebeneffekte.

Am Beispielnetz in Abbildung 1.4 (S. 10) kann man sich klar machen, dass eine Erweiterung des Lokalitätsbegriffs auf Nebeneffekte der Anschriftsprache dem Zweck des Lokalitätsprinzips vollkommen zuwider laufen würde:

Bei Netzexemplaren, synchronen Kanälen und Guards lässt sich noch an der grafischen oder formalen Darstellung aller beteiligten Netzmuster die Lokalität eines Elements erkennen.

Bei den pathologischen Beispielen in den Abbildungen 1.3 und 1.4 muss man aber die Eigenheiten der verwendeten Typen und Operatoren kennen, um das Verhalten des Netzes zu verstehen und abzuschätzen, ob irgendwo in einem der Methodenaufrufe ein Nebeneffekt auftreten kann, der die Aktivierung einer Transition beeinflusst.

Eine Erweiterung des Lokalitätsbegriffs um Nebeneffekte würde also nicht mehr mit dem intuitiven Verständnis[10] von Lokalität zusammenfallen, da die Objekte nicht mehr durch Elemente des Netz-

[10] Den Begriff des „intuitiven Verständnisses“ eines Begriffs habe ich aus der analytischen Philosophie entlehnt. Die Bezeichnung „intuitiv“ für „vor-theoretisch“ oder „dem Alltagsverständnis entsprechend“ wird dort sehr häufig verwendet. Eine exakte Definition wird man aber aus offensichtlichen Gründen niemals finden. Diskussion und Kritik siehe z. B. [Craig, 1993].

formalismus selbst verbunden (quasi „am gleichen Ort“), sondern durch die Semantik des verwendeten Typsystems miteinander verknüpft sind.

Auch wäre der Begriff in einer solchen Erweiterung nicht mehr so nützlich um über Nebenläufigkeit und Synchronisation nachzudenken, da die Lokalitätsbeziehungen bedeutend schwieriger zu erkennen wären.

Ein Vorteil von Petrinetzen besteht aber gerade darin, dass sich mit Ihnen Nebenläufigkeit und Synchronisation gut darstellen und verstehen lassen, was durch solche Effekte verhindert wird.

Eine Erweiterung des Lokalitätsbegriffs auf Nebeneffekte einer Anschriftsprache erscheint darum generell nicht sinnvoll.

Sofern eine Anschriftsprache Nebeneffekte ermöglicht, werden diese potentiell das Prinzip der Lokalität verletzen.

Der hier festgelegte Lokalitätsbegriff ist also zunächst nur für Referenznetze ohne Berücksichtigung der Anschriftsprache nützlich.

Ein weiterer Vorteil des vorgeschlagenen Lokalitätsbegriffs besteht darin, dass er lediglich auf in [Kummer, 2002] formal definierte Eigenschaften von Referenznetzen Bezug nimmt, einer formalen Behandlung also nichts im Wege steht.

Allerdings ist der vorgeschlagene Lokalitätsbegriff für Referenznetze noch etwas unterspezifiziert. Verschiedene, in Referenznetzen mögliche, Kombinationen von speziellen Kantentypen (etwa Test- oder Inhibitor-Kanten) haben zum Teil subtile Auswirkungen auf die Nebenläufigkeitssemantik (siehe etwa [Baldan u. a., 2000] oder auch [Kummer, 2002] selbst), die durch eine Fallunterscheidung für verschiedene Kantentypen in der Definition auf Seite 15 abgehandelt werden könnten um zu einem allgemeinen und präzisen Begriff zu gelangen.

Auch wäre es bedenkenswert, ob wie bei den Netzen der CPN TOOLS die Lokalität von der Markierung abhängen sollte, so dass zwei Transitionen nebenläufig Marken aus der gleichen Stelle entfernen könnten, sofern die Multimengen der gebundenen Marken disjunkt sind (siehe [Kristensen und Christensen, 2004]).

Trotz ihrer Grobheit ist die Definition für diese Diskussion ausreichend, weil ich in den Beispielnetzen zur Vereinfachung bewußt auf diese Kantentypen verzichtet habe. Es werden lediglich Reservekanten als Abkürzungen für verdoppelte gegenläufige Kanten mit gleicher Beschriftung verwendet.

Welche Probleme ergeben sich?

Ich habe eine Festlegung des Lokalitätsbegriffs für Referenznetze vorgeschlagen, die die Anschriftsprache ausklammert, denn die durch Java möglichen Nebeneffekte lassen sich weder einfach kontrollieren noch verhindern. Die Einschränkungen der Java-Anschriften gegenüber dem vollen Sprachumfang setzen der Erzeugung von nicht-lokalen Effekten kaum nennenswerte Hindernisse entgegen.

In RENEW war von vornherein das Hauptaugenmerk auf Simulation gerichtet und „Analyse blieb fast vollständig unberücksichtigt“ [Kummer, 2002, S.233]. Auch lassen sich mit RENEW tatsächlich komplette Anwendungen programmieren. RENEW wurde als eine Art Entwicklungsumgebung für Petrinetze konzipiert. Die Netze werden in der Simulation getestet und können mit Funktionen untersucht werden, die denen eines (rudimentären) Debuggers entsprechen.

In diesem Anwendungsbereich sind die beschriebenen Nebeneffekte nicht prinzipiell problematisch, wenngleich sie eine analytische Betrachtung der erstellten Systeme erschweren können.

In meiner Studienarbeit [Delgado Friedrichs, 2006] habe ich z. B. argumentiert (Bezug nehmend auf [Mäkelä, 2001] und [Mäkelä, 2003]), dass Optimierungstechniken für Model-Checking-Verfahren durch solche Effekte stark erschwert werden. Auch die in den CPN TOOLS implementierten Analyseverfahren und Optimierungen setzen die Einhaltung des Lokalitätsprinzips voraus.

Im RENEW-Handbuch [Kummer u. a., 2006b] wird verschiedentlich empfohlen, Synchronisation und Nebenläufigkeit durch Netze zu modellieren und von Synchronisationsmechanismen und anderen Nebeneffekten der Anschriftsprache Abstand zu nehmen. Für die

Entwicklung von Anwendungen ist eine solche Empfehlung auch vollkommen ausreichend. Für die Verifikation mittels automatischer Techniken wie etwa Model-Checking oder auch für Beweisverfahren wäre es vorteilhaft, wenn die Abwesenheit von Nebeneffekten *garantiert* werden könnte, da davon abhängen kann, ob die jeweiligen Verfahren korrekte Ergebnisse liefern.

Des Weiteren konnte ich an den Beispielen zeigen, dass auch die informelle Betrachtung von Referenznetzgruppen durch die Verwendung von Nebeneffekten stark erschwert werden kann. Der Programmierer von verwendeten Java-Methoden ist angehalten, bei der Implementierung seiner Klassen sehr diszipliniert vorzugehen, weil diverse Eigenschaften der Anschriftsprache zu Problemen führen können, die schwer zu erkennen und zu beheben sind. Einige *einfache* Beispiele für Probleme dieser Art habe ich in diesem Abschnitt ausführlich diskutiert, andere nur angedeutet (wie etwa die Problematik von synchronisierten Java-Methoden zum Netz in Abbildung 1.4 (S. 10)).

Die Beispiele zeigen außerdem deutlich, dass die syntaktischen Einschränkungen der Java-Beschriftungen kaum für eine wirksame Beschränkung der Ausdrucksmöglichkeiten sorgen.

Der im Kapitel „Auswertung von Ausdrücken in Scheme-Referenznetzen“ (S.59) gegebene Mechanismus soll darum Einschränkungen auf semantischer Ebene erlauben, mit denen sich Nebeneffekte ausschließen lassen, da gezeigt wurde, dass die Anwesenheit von Nebeneffekten sinnvolle Vorhersagen über das Verhalten eines Netzes selbst in trivialen Fällen stark erschwert.

Mit Scheme als Beschriftungssprache können Nebeneffekte besser vermieden werden als mit Java, da ein rein funktionaler und ausdrucksmächtiger Kern der Sprache existiert.

2 Motivation und Ziel der Arbeit

Ziel der Arbeit ist die Konzeption und Implementation einer möglichst engen Integration von Scheme und Referenznetzen im Rahmen der Anwendung RENEW.

Es soll begründet werden, warum Scheme gut als Beschriftungssprache für Referenznetze geeignet ist.

Die Scheme-Beschriftungen sollen alle Möglichkeiten von Java-Beschriftungen bieten und die Vorteile der Programmiersprache Scheme in Referenznetzen nutzbar machen.

Außerdem soll bereits in der Konzeption des neuen RENEW-Formalismus ein Mechanismus angeboten werden, mit dem sich die in Abschnitt 1.3.4 (S. 12) angesprochenen Probleme umgehen lassen, um daran einen konkreten Vorteil der neuen Beschriftungssprache gegenüber Java praktisch verdeutlichen zu können.

Die Problemlösungen der Implementation, die sich aus der Implementation ergebenden Konventionen der Anschriftsprache und die durch das Plugin angebotenen Dienste sollen, so weit dies in einer in Java geschriebenen Anwendung möglich ist, den Gepflogenheiten und Erwartungen von Scheme-Programmierern entgegenkommen.

Dies bedeutet zum Beispiel, dass eine Möglichkeit zum interaktiven Testen und Debuggen angeboten wird, wie sie in Lisp-Systemen üblich ist, dass lokale Definitionen verwendet werden können und dass Prozeduren und andere Scheme-Objekte erster Klasse als Markierungen in Petrinetzen zugelassen sein sollen. Überall dort, wo sich die üblichen Verfahrensweisen bei Scheme von denen bei Java unterscheiden, sollen die ersteren bevorzugt werden.

Gleichzeitig sollen aber so weit wie möglich bestehende Programmteile von RENEW genutzt werden. Dieses Ziel steht mit dem

vorigen in Konflikt. Insbesondere beim Parsing und bei der Unifikation erscheint es wenig sinnvoll dies mit den Mitteln der Sprache Java zu versuchen, da in Scheme dafür bessere Methoden zur Verfügung stehen; andererseits wäre es wenig sinnvoll, einen eigenen vollständigen Simulator in Scheme zu schreiben. Es soll darum versucht werden, die Scheme-Programmteile so weit mit den existierenden Java-Programmteilen zu integrieren, dass es zu möglichst wenig Verdoppelung von Funktionalität kommt.

Dass ein eigenes Unifikationsverfahren verwendet wird, ist dadurch gerechtfertigt, dass der Scheme-Formalismus sich von allen bisher in RENEW implementierten Netzformalismen so stark unterscheidet, dass es ohnehin nötig gewesen wäre, den bestehenden Unifikationsalgorithmus sehr stark anzupassen.

Ein zusätzliches Ziel der Implementation besteht darin, ausschließlich solche Bibliotheken für Scheme und/oder Java zu verwenden, die mit der Lizenz von RENEW verträglich sind, damit das fertige Plugin als Bestandteil der Software weiterhin unter einer Open-Source Lizenz verbreitet werden kann.

3 Arbeiten zur Thematik

Eine Recherche nach Arbeiten zu Petrinetzen in Verbindung mit der Programmiersprache Scheme lieferte keine Ergebnisse. Anscheinend gibt es zu dieser konkreten Kombination bisher zumindest keine bekannten Arbeiten.

Interessant sind in diesem Zusammenhang nur Arbeiten, die gefärbte Netze behandeln, im Gegensatz zur bloßen Implementation von Simulatoren oder Editoren für einfache Petrinetze mit Lisp oder anderen funktionalen Sprachen.

3.1 CPN Tools

Wird das Blickfeld auf funktionale Sprachen erweitert, findet sich zunächst die Beschreibung des Tools DESIGN/CPN [Varhol, 1991], welches von den CPN TOOLS der CPN Gruppe der Universität Aarhus abgelöst wurde. Aus der Gruppe um Kurt Jensen gibt es etliche Publikationen zum Thema, die einen großen Teil der verfügbaren Artikel zum Thema funktionale Sprachen und gefärbte Netze darstellen dürften. Insbesondere [Kristensen und Christensen, 2004] gibt Einblicke in die Techniken der Implementation.

Der dort beschriebene Algorithmus zur Bindungssuche beruht auf einer eingeschränkten Variante des Pattern-Matchings der verwendeten Sprache Standard-ML.

Durch das Typsystem von SML werden Optimierungen sowohl bei der Bindungssuche als auch der Zustandsraumanalyse möglich, welche die CPN TOOLS von vornherein unterstützen.

Der Algorithmus zur Bindungssuche ist, wenn man von den Details des Pattern-Matching bzw. der Unifikation absieht, dem von RENEW nicht unähnlich. In beiden Fällen werden Ausdrücke erst

auf Typkompatibilität und Abhängigkeiten analysiert, so dass z. B. in beiden Programmen Guards zum Teil ausgewertet werden können, bevor die Bindungen ganz vollständig sind.

3.2 Haskell-Coloured Petri Nets

Die rein funktionale Programmiersprache Haskell wird in [Reinke, 2000] mit gefärbten Petrinetzen verknüpft.

Dieser Artikel gibt eine ausgesprochen knappe und elegante Einbettung von gefärbten Petrinetzen in die Programmiersprache Haskell. Die zur Verfügung stehenden Patterns sind aber gegenüber denen der Referenznetze oder CPN-Netze deutlich eingeschränkt. Transitionen werden explizit als Haskell-Funktionen spezifiziert, die Markierungen auf Listen von Nachfolgemarkierungen abbilden, und das Netz wird dann aus Transitionen aufgebaut. Dadurch kann bei den Transitionen zwar ebenfalls das Pattern-Matching von Haskell zur Verwendung kommen, dies ist aber lokal zum Teilausdruck an der jeweiligen „Kante“, so dass bei dieser Art der Einbettung nicht gleiche Objekte an unterschiedlichen Kanten durch die Verwendung des gleichen Variablennamens gebunden werden können (wie etwa im Beispiel 8.3 (S. 99)).

Eine Einbettung in Scheme könnte darüber hinaus eigene Syntax zur Spezifikation von Netzelementen bereitstellen, so dass die textuelle Beschreibung noch stärker die Struktur des Netzes widerspiegelt, ähnlich wie dies etwa beim Model-Checking-Tool MARIA der Fall ist.

3.3 Bewertung

Die gesichtete Literatur, inklusive der Dissertation „Referenznetze“ von Olaf Kummer ([Kummer, 2002]) erweckt insgesamt den Eindruck, dass die Bindungssuche das Hauptproblem bei der Implementation von gefärbten Netzen darstellt. Bei Kummer kommen Referenznetzexemplare und bei Mäkelä (u.a. [Mäkelä, 2002]) sowie

[Kristensen und Christensen, 2004] die Zustandsraumexplosion als weitere Schwierigkeiten hinzu.

Die bei Model-Checking Verfahren auftretende Explosion des Zustandsraums ist für diese Arbeit nicht direkt ein Thema. Der implementierte Einschränkungsmechanismus sollte aber auch nutzbar sein, um z. B. Erreichbarkeitsgraphen etwas zu vereinfachen.

Die von Reinke vorgestellte Einbettung in Haskell zeigt eine für die Zukunft sehr interessante alternative Implementierung oder auch Erweiterung einer bestehenden Implementierung von Scheme-Referenznetzen auf.

Teil II

Konzeption

4 Alternative Beschriftungssprachen für Referenznetze

Dieses Kapitel entwickelt die Anforderungen an die neue Beschriftungssprache.

Der erste Abschnitt untersucht allgemeine Kriterien für RENEW-Beschriftungssprachen. Anschließend werden andere funktionale Sprachen als Alternativen zu Scheme untersucht. Der letzte Abschnitt wählt die Implementation SISC [SISC, 2006] unter einigen Kandidaten aus, untersucht anhand eines Kriterienkataloges aus [Kummer, 2002] die Eignung dieser Implementation und fasst die daraus gewonnenen konkreten Anforderungen zusammen.

4.1 Kriterien

Eine Anschriftsprache für RENEW, die es ermöglicht die im Abschnitt 1.3.4 (S. 12) angeführten Probleme zu vermeiden oder zu umgehen, sollte die folgenden Kriterien erfüllen:

1. Gute Integration mit Java.

   Eine absolute Minimalanforderung ist hierbei, dass Ausdrücke in der Programmiersprache direkt aus dem Java Laufzeitsystem heraus ausgewertet werden können *müssen*.

   Weiterhin wäre es *wünschenswert* wenn schon die Implementation des RENEW-Plugins selbst in der betreffenden Sprache erfolgen kann, damit die Anwender es so weit wie möglich in ihrer *bevorzugten* Sprache weiterentwickeln können. Weil

der Kern von RENEW in Java geschrieben ist, muss darum aus der Beschriftungssprache heraus Zugriff auf Java-Objekte der laufenden virtuellen Maschine möglich sein.

2. Da der RENEW Simulator mehrere nebenläufige Threads („Ausführungsfäden“) verwendet, muss die verwendete Sprachimplementation robust gegenüber nebenläufiger Ausführung sein, bzw. diese unterstützen.

3. Weiterhin wird ein Mechanismus benötigt, mit dem die in Abschnitt 1.3 (S. 7) beschriebenen Nebeneffekte wirksam vermieden werden können, damit die Einhaltung des Lokalitätsprinzips unter noch genauer zu spezifizierenden Voraussetzungen gewährleistet und Garantien für eine leichtere informelle Abschätzung des Netzverhaltens gegeben werden können.

   Wenn ein solcher Mechanismus zur Verfügung steht, sind noch nicht alle Kriterien erfüllt, die zur Implementation eines optimierten Model-Checking-Algorithmus oder anderer Analyseverfahren notwendig sind. [Mäkelä, 2001] und [Mäkelä, 2003] geben z. B. Datentypen mit eingeschränktem und streng geordnetem Wertebereich und weitere Bedingungen für optimierte, auf Entfaltung basierende, Model-Checking-Verfahren vor.

   Vorgaben wie diese werden sich jedoch für Scheme nur schwer umsetzen lassen. Die polymorphe Liste z. B. kann als Grunddatentyp der Sprache wohl nur mit sehr großem Aufwand und deutlichen Einschränkungen ihrer Ausdrucksmächtigkeit auf einen eingeschränkten, streng geordneten Wertebereich abgebildet werden (wobei die Ordnung auch nicht auf plattformspezifischen Eigenschaften beruhen darf).

   Der konkrete Mechanismus wird auf der Kontrolle der sichtbaren Bindungen basieren, wodurch sich sichtbare Nebeneffekte verhindern lassen, indem die Bindungen, mit denen

Nebeneffekte erzeugt werden können, entfernt oder umdefiniert werden. Aber eine einschneidende Änderung an einem so grundlegenden Datentyp wie der Liste in Scheme wird sich damit hingegen kaum sinnvoll umsetzen lassen.

Model-Checking beispielsweise wird also kaum leichter für Scheme-Referenznetze als für Java-Referenznetze realisierbar werden. Vielleicht lassen sich aber durch Nebeneffektfreiheit oder andere durch den Mechanismus mögliche Einschränkungen schon andere Analyseverfahren leichter implementieren, oder gewisse Zusicherungen über das Verhalten der Netze lassen sich leichter einhalten.

Obwohl schon durch das Thema feststeht, dass in dieser Arbeit Scheme verwendet wird, sollen dennoch im Folgenden einige wenige Alternativen untersucht werden.

4.2 Rein funktionale Sprachen

Eine zunächst offensichtlich erscheinende Lösung wäre die Einführung einer rein funktionalen Anschriftsprache für Referenznetze.

Was der Begriff der „rein funktionalen Programmiersprachen“ genau bezeichnet ist nicht unumstritten, in der Regel werden die Sprachen Haskell und Miranda als „rein funktional“ bezeichnet, während Standard-ML und Scheme nicht als rein bezeichnet werden, weil sie Nebeneffekte (insbesondere Zustandsänderungen) erlauben[1].

rein funktionale Programmiersprache

Mit **rein funktionalen Programmiersprachen** sind darum in dieser Arbeit *vereinfachend* solche gemeint, die nur rein lokale Nebeneffekte erlauben. D.h. Nebeneffekte sind auf einen lokalen Gültigkeitsbereich begrenzt und sind außerhalb dessen nicht beobachtbar, was z. B. durch die Monaden in Haskell erreicht wird. Es kann

[1] Eine strenge formale Definition des Begriffs findet sich in [Sabry, 1998]. Dort wird aber nicht einmal (die damalige Implementation von) Haskell als völlig rein bezeichnet.

insbesondere kein global sichtbarer Zustand verändert werden. Damit sind auch die problematischen Fälle aus dem vorigen Abschnitt automatisch ausgeschlossen.

Zwei Kandidaten für in diesem Sinne rein funktionale Beschriftungssprachen werden im Folgenden vorgestellt.

4.2.1 MARIA

Die Petrinetz-Beschreibungssprache des Model-Checkers MARIA von Marko Mäkelä [Mäkelä u. a., 2005] erlaubt über die Veränderung der Markierung hinaus keinerlei Zustandsänderungen.

Vorteile dieser Sprache bestehen zum Einen darin, dass sie als Petrinetzsprache konzipiert wurde und alle Sprachkonstrukte darum gut zu Semantik von Petrinetzen passen. Die zugrundeliegenden algebraischen Netze [Kindler und Völzer, 1997] haben gewisse Gemeinsamkeiten mit Referenznetzen. Eine Anpassung an RENEW und Referenznetze wäre also durchaus denkbar.

Zum Anderen wurde diese Sprache bewusst mit dem Ziel entworfen, Optimierungen von Model-Checking Verfahren zu erleichtern. Es wäre also plausibel anzunehmen, dass andere Analyseverfahren ebenfalls erleichtert würden. (Insbesondere wurden Nebeneffekte ausgeschlossen und alle Datentypen haben beschränkte Wertebereiche und sind (architektur- und implementationsunabhängig) streng geordnet.)

Es existiert bereits ein Plugin, mit dem einfache MARIA-Netze in RENEW erstellt und exportiert werden können und das außerdem eine Oberfläche für die Bedienung des Model-Checkers zur Verfügung stellt. Die Simulation von MARIA-Netzen könnte zusätzlich noch in Java nachempfunden werden; wie jedoch bereits in [Delgado Friedrichs, 2006] diskutiert erscheint es mir bedeutend einfacher, für die Simulation von MARIA-Netzen die bereits vorhandene Kommunikation mit dem Model-Checker zu erweitern. Die Implementation dieser Erweiterungen kann aufbauend auf der bisherigen Arbeit mit wenig theoretischer und konzeptioneller Arbeit durchgeführt werden und erschien mir darum nicht als lohnendes Diplom-

arbeitsthema.

Außerdem ist die MARIA-Sprache keine Allzweck-Programmiersprache (im Gegensatz zu Java, Haskell oder Scheme). Ihre Ausdrucksmächtigkeit beschränkt sich auf die Spezifikation und Verifikation von Modellen, wie z. B. Netzwerkprotokollen. Benutzereingaben und Ausgaben können lediglich simuliert werden. Damit lassen sich mit MARIA auch in Verbindung mit RENEW keine echten Anwendungen programmieren. Die Integration des Model-Checkers wird also mit einer deutlichen Einschränkung erkauft.

4.2.2 Haskell

Die Sprache Haskell ist eine rein funktionale Sprache mit träger Auswertung[2] und statischen, polymorphen Typen [Peyton Jones, 2003]. Ein besonderes Ausdrucksmittel von Haskell sind Monaden, mit denen Nebeneffekte so gekapselt werden können, dass sowohl die Reihenfolge ihres Auftretens garantiert werden kann, als auch die positiven Eigenschaften der trägen Auswertung erhalten bleiben (wie u.a. in [Launchbury, 1993] u. v. a. diskutiert).

Haskell ist eine Allzwecksprache. Ein- und Ausgabeoperationen wurden so integriert, dass jede beliebige Anwendung implementiert werden kann.

Damit bliebe bei einer Integration von Haskell die Zielsetzung von RENEW erhalten. Zusätzlich dürfte die Lösung der hier angesprochenen Probleme auf Basis eines Haskell-Plugins für RENEW leicht gelingen. Es gibt zwar noch keine formale Spezifikation der Haskell Semantik, aber dafür erlaubt Haskell auf Grund der weitgehenden Beachtung von referentieller Transparenz, Definitheit und Entfaltbarkeit[3] sehr einfache Beweise durch Rechnen mit Programmen (Beispiele dafür gibt [Gibbons, 2002]).

Außerdem bietet die von Reinke (in [Reinke, 2000]) vorgestellte

[2] engl.: „lazy evaluation“. Im deutschen wird auch „Bedarfsauswertung“ verwendet („call-by-need“) was ebenfalls eine gute Entsprechung ist, oder auch „verzögerte Auswertung“ („delayed evaluation“).

[3] Begriffsdefinitionen siehe [Søndergaard und Sestoft, 1990]

Einbettung in Haskell bereits einen Petrinetzformalismus auf Haskell-Basis.

Zwei Gründe sprechen dennoch dagegen, Haskell in dieser Arbeit an Stelle von Scheme zu verwenden:

1. Der erste Grund ist objektiver Natur:

   Es gibt noch keine so vollständige und ausgereifte Integration eines Haskell-Interpreters oder Compilers in Java, wie das bei Scheme der Fall ist. Die existierenden Projekte werden nicht mehr aktiv entwickelt oder ermöglichen (noch) keine vollständige Integration (nur Aufruf von Haskell aus Java oder Übersetzung von Haskell nach Java Bytecode) oder befinden sich noch in einem frühen Stadium der Konzeption:

   Zur Verdeutlichung folgt eine Auflistung der drei vielversprechendsten Ansätze, zu denen es auch eine Implementation (oder Komponenten einer solchen) gibt.

   Ein Ansatz zur vollständigen Integration wird in [Meijer und Finne, 2001] beschrieben, aber kein Konzept zur Interpretation von Haskell-Code zur Laufzeit einer Java-VM. Außerdem scheint das Projekt nicht mehr gepflegt zu werden und es gibt keine Information über Lizenz und Verfügbarkeit von Quellcode.

   JASKELL [Vernet, 1998] ist ein Compiler für Haskell nach Java-Bytecode. Ein ähnlicher Ansatz ließe sich möglicherweise mit einer Anpassung oder Erweiterung von YHC [Yhc, 2006] verfolgen. Eine Literaturrecherche förderte weitere Arbeiten zur Compilation von Haskell nach Java-Bytecode zu Tage.

   Es zeigt sich also, dass das Kriterium der Java-Integration das härtere der auf Seite 29 genannten ist.

   Zusätzlich müsste die verwendete Sprachimplementation robust gegenüber Nebenläufigkeit sein. Von den genannten Kandidaten ist nur bei dem Compiler YHC bekannt, dass er Nebenläufigkeit unterstützt. Dies könnte für die anderen genannten Kandidaten ein weiteres Problem darstellen.

In Abwesenheit einer Java-Integration von Haskell könnte die von Reinke vorgestellte Einbettung mit Referenznetzexemplaren ergänzt und mit ähnlichen Techniken wie MARIA zur Zusammenarbeit mit RENEW bewegt werden. Die „Haskell-Coloured Petri Nets“, wären damit aber kein integraler Bestandteil von RENEW. Ähnlich wie bei dem MARIA-Plugin wäre ihre Verwendung mit zusätzlicher Arbeit für den Anwender verbunden, weil eine Haskell-Laufzeitumgebung bzw. ein Compiler installiert werden müsste.

Damit sind die Hauptvertreter der reinen funktionalen Sprachen ausgeschieden. Für die funktionalen Programmiersprachen Clean und Miranda gibt es nicht ansatzweise eine Integration in Java, Miranda steht noch dazu aus Lizenzgründen für den Einsatz mit RENEW außer Frage.

2. Der subjektive Grund besteht darin, dass ich persönlich Scheme deutlich vor Haskell, bzw. allgemeiner nicht-rein funktionale Sprachen vor reinen bevorzuge.

   Der wesentliche Vorteil von rein funktionalen Sprachen wie Haskell besteht darin, dass alle Ausdrücke implizit träge ausgewertet werden, was eine sehr elegante Notation erlaubt und in Sprachen mit Nebeneffekten, wenn überhaupt, nur sehr umständlich zu realisieren wäre (so wird z. B. in [Launchbury, 1993] argumentiert).

   Dafür bieten nicht-rein funktionale Sprachen alle *anderen* in [Hughes, 1989] diskutierten Vorteile der funktionalen Programmierung und darüber hinaus alle Vorteile (und Versuchungen) der imperativen Programmierung.

   Außerdem lässt sich in allen strikten, funktionalen Sprachen leicht eine explizite Form der verzögerten Auswertung implementieren, wie sie z. B. im Scheme-Standard [R^{5}RS, 1998] enthalten ist und in den `SRFI`-Dokumenten [SRFI-40, 2004] und [SRFI-45, 2004] erweitert wird.

   In [Søndergaard und Sestoft, 1990] wird demonstriert, dass

bestimmte Eigenschaften von Programmiersprachen sich einander sogar formal ausschließen können. Implizit träge Auswertung und Nebeneffekte sind Ausdrucksmittel, die sich (zumindest praktisch) gegenseitig ausschließen, also muss zwischen ihnen gewählt werden. Da keine zwingend notwendigen Gründe für eine der beiden Seiten und gegen die andere sprechen, handelt es sich also um eine Frage des Stils.

4.3 Funktionale Sprachen

Der Grund, warum jetzt nur noch Sprachen in Erwägung gezogen werden können, die funktionale Programmierung unterstützen, ist das Kriterium 3 (S. 30). Wenn ein Mechanismus verwendet werden soll, der Nebeneffekte aus der Sprache entfernt, dann sollte die Sprache weiterhin so ausdrucksmächtig bleiben, dass sich interessante Aufgaben damit lösen lassen. Bei funktionalen Sprachen ist dies durch Funktionen höherer Ordnung, „Continuation Passing Style“, Monaden oder ähnliche Ausdrucksmittel gegeben.

Es wäre aus diesem Grund ausgesprochen schwierig, z. B. Java auf eine nebeneffektfreie Untermenge einzuschränken.

Eine bedenkenswerte Alternative zu Scheme wäre hier Ruby, das zwar nicht als funktionale Sprache im herkömmlichen Sinne bezeichnet wird, aber alle strikten Techniken der funktionalen Programmierung unterstützt. Es gibt mit JRUBY[4] eine *fast* vollständige Ruby Implementation in Java mit vollständiger Integration von Java. Darüber hinaus nimmt die Ruby-Methode `Kernel.eval()` als zweiten Parameter ein Objekt der Klasse `Binding`, was als Umgebung des auszuwertenden Ausdrucks verwendet werden kann. Einen ganz ähnlichen Mechanismus wird das Scheme-Plugin verwenden um das Kriterium 3 in der auf Seite 30 gegebenen Liste zu erfüllen. Leider lassen sich mit diesem Mechanismus keine Methoden aus dem `Binding` Objekt entfernen, sondern nur aus der Klasse. Das Binding-Objekt bietet also keine so umfassenden Manipulati-

[4] http://www.jruby.org/

onsmöglichkeiten wie die Umgebungen der Scheme-Implementation SISC (siehe Abschnitt 4.4.1 (S. 40)).

Möglicherweise wäre die Verwendung von `taint` und `$SAFE` für die Auswertung von Beschriftungen ein besserer (ruby-gemäßerer) Ansatz.

Es zeigt sich, dass Scheme sowohl auf Grund der guten Java-Integration durch SISC, als auch durch die Möglichkeit, den Ausführungskontext explizit zu kontrollieren, unter den untersuchten Alternativen die beste ist.

4.4 Die Programmiersprache Scheme

Ich zitiere die Spracheigenschaften aus [R^{5}RS, 1998], der aktuellen fünften Revision des Sprachstandards (ab jetzt R^{5}RS abgekürzt). Die sechste Revision (R^{6}RS) ist am 12. August 2007 mit knapper Mehrheit ratifiziert worden, aber immer noch sehr stark umstritten. Es gibt neben den von der Scheme-Gemeinde gemeinschaftlich entwickelten „revised“ Standards auch einen offiziellen IEEE Standard ([IEEE Scheme, 1991]), zu dem R^{5}RS eine echte Obermenge bildet. Der IEEE Standard ist heute praktisch ohne Bedeutung, da es nur sehr wenige Implementationen gibt, die sich auf ihn beschränken. (Allerdings bezieht sich das bedeutende Informatiklehrbuch [Abelson u. a., 1985] auf diesen Standard.) Faktisch bildet er eine Zwischenstufe zwischen der vierten und der fünften Revision des Community-Standards.

Scheme ist ein Dialekt von Lisp mit lexikalischem Gültigkeitsbereich („static scoping“), dynamischen Typen und Unterstützung für Endrekursion. Typen sind mit Werten (auch: Objekten) assoziiert (und nicht mit Variablen). Alle erzeugten Objekte haben unendlich lange Lebensdauer; wenn ein Objekt nicht mehr referenziert werden kann, wird es durch den „Garbage Collector“ entfernt. Scheme

Besondere Datentypen in Scheme sind **Closures** (Prozeduren mit eigener lexikalischer Umgebung) und **Umgebungen** („Environments“). Closures Umgebungen

[R5RS, 1998] charakterisiert „Umgebung“ wie folgt[5]:

Bezeichner

- Ein **Bezeichner** kann eine Art von Syntax[6] bezeichnen oder einen Platz an dem ein Wert gespeichert werden kann.

syntaktisches Schlüsselwort

- Ein Bezeichner, der eine Art von Syntax bezeichnet, wird **syntaktisches Schlüsselwort** genannt und ist an die Syntax „*gebunden*“.

Variable

- Ein Bezeichner, der einen Platz bezeichnet, wird **Variable** genannt und ist an den Platz „*gebunden*“.

Umgebung

- Die Menge aller sichtbaren Bindungen, die zu einem Ausführungszeitpunkt eines Programmes wirksam ist, wird als die zum Ausführungszeitpunkt wirksame **Umgebung** bezeichnet.

Umgebungen spielen eine bedeutende Rolle in der in [R5RS, 1998] spezifizierten denotationalen Semantik.

Im Kapitel 6 (S. 59) wird ein Verfahren konzipiert, mit dem die im Unterabschnitt 1.3.4 (S. 12) diskutierten Probleme bei Scheme-Anschriften gezielt umgangen werden können. Das Verfahren nutzt Scheme-Umgebungen um bestimmte Bindungen gezielt unsichtbar zu machen und so Nebeneffekte einzuschränken.

Continuations

„**Continuations**“ (Fortsetzungen) wiederum sind spezielle Closures. Zu jedem Ausführungszeitpunkt existiert die Fortsetzung der aktuellen Berechnung. Es handelt sich dabei um eine Closure, also eine Prozedur, die als Argument den aktuellen Stand der Berechnung erhält und daraus den vollständigen Wert der Berechnung bestimmt. (Bei Closures die mehrere Werte liefern, kann es sich auch

[5] Originaltext: An *identifier* may name a type of syntax, or it may name a location where a value can be stored. An identifier that names a type of syntax is called a *syntactic keyword* and is said to be *bound* to that syntax. An identifier that names a location is called a *variable* and is said to be *bound* to that location.The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point.

[6] Hiermit sind die „special forms“ und benutzerdefinierte Makros gemeint, die nach einem anderen Verfahren ausgewertet werden als Prozeduranwendungen.

um Continuations mit mehreren Argumenten handeln.) Mit einer speziellen Prozedur `call-with-current-continuation` (oder auch `call/cc`) lässt sich die aktuelle Continuation festhalten und die aktuelle Berechnung lässt sich mit Hilfe der eingefangenen Continuation abbrechen und zu einem beliebigen späteren Zeitpunkt, mit beliebigen Argumenten, so oft wie gewünscht, erneut aufrufen.[7]

Scheme Prozeduren, Continuations und Umgebungen sind Objekte erster Klasse (Die Möglichkeit der Modifikation von Umgebungen ist in reinem R^{5}RS-Scheme aber nicht spezifiziert. Die Scheme-Implementation SISC bietet erweiterte Möglichkeiten für die Arbeit mit Umgebungen).

Es gibt in Scheme zwei ausgezeichnete Boolean-Werte, nämlich `#f` für „falsch“ und `#t` (true) für „wahr“. In Verzweigungen des Kontrollflusses wird aber *ausschließlich* der Wert `#f` als boolescher Wert „falsch“ interpretiert. Es gilt:

`(not wert)` liefert `#t` falls `(eq? wert #f)`, ansonsten `#f` und

`(if wert #t #f)` liefert `#f` falls `(eq? wert #f)`, ansonsten `#t`.

Weiterhin verfügt Scheme über ein hygienisches Makrosystem, mit dem neue syntaktische Konstrukte zur Sprache hinzugefügt werden können. Als „hygienisch“ wird die Eigenschaft bezeichnet, dass in der Expansion des Makros nicht unabsichtlich Variablen aus dem umgebenden Gültigkeitsbereich referenziert werden können, was als „variable capture“ eine bekannte Fehlerquelle bei einfacheren Makrosystemen (wie etwa dem von ANSI Common LISP) bildet. ([Kiselyov, 2002] zeigt jedoch Schwächen der in [R^{5}RS, 1998] spezifizierten Hygiene auf.)

Ein wichtiger Unterschied zwischen Scheme und Java besteht darin, dass für Java nur verschiedene Implementationen von Compilern und Laufzeitumgebungen (SUN JDK/JRE, BLACKDOWN, KAFFE) gibt, die aber (abhängig von der Version) im wesentlichen den gleichen Sprachumfang bereitstellen.

[7] Siehe ebenfalls [R^{5}RS, 1998] für eine knappe Erläuterung und Beispiele. Eine verständliche Einführung für schon etwas fortgeschrittene Schemer gibt [Friedman, 1988]

4.4.1 Auswahl einer Scheme-Implementation

Der Scheme-Standard in der aktuellen Ausprägung R^{5}RS ist dagegen ausgesprochen minimalistisch. Insbesondere werden keine Möglichkeiten definiert, auf die Hardware oder das Betriebssystem zuzugreifen, demzufolge definiert der Standard auch keine Möglichkeit des Zugriffs auf Dateien oder auf Kommunikationsnetzwerke. Solche Details werden der Implementation überlassen.

Aus diesem Grund gibt es eine Vielzahl von R^{5}RS-kompatiblen Scheme-Implementationen, die sich in den Details zum Teil sehr stark voneinander unterscheiden.

Auf der Webseite http://community.schemewiki.org/?scheme-faq-standards#implementations wurden im September 2007 immerhin 68 verschiedene aufgezählt, die möglichen Kandidaten sind jedoch für mein Anliegen leicht einzugrenzen:

Acht der aufgelisteten Implementationen sind in Java geschrieben, was eine der bereits genannten Voraussetzungen für eine nahtlose Integration in RENEW ist, da der Benutzer möglichst nicht auf einen externen Compiler oder Interpreter angewiesen sein soll.

Gesucht ist eine Scheme-Implementation in Java, die Zugriff auf Java-Objekte von Scheme-Code aus und Zugriff auf Scheme von Java-Code aus bietet, was ich bereits als „**vollständige Integration von Scheme und Java**“ bezeichnet habe. Außerdem soll der ganze R^{5}RS-Standard unterstützt werden und die Implementation soll möglichst ausgereift sein und noch aktiv weiterentwickelt werden. Eine zusätzliche, bereits genannte Anforderung besteht darin, dass die Implementation thread-safe sein muss.

vollständige Integration von Scheme und Java

JSCHEME (http://jscheme.sourceforge.net/jscheme/main.html) Entspricht lediglich der nicht mehr aktuellen vierten Revision des Standards (R^{4}RS).

JAJA (http://www-spi.lip6.fr/~queinnec/Java/Jaja.html) wird nicht mehr aktiv entwickelt und das Nachfolgeprojekt PS^{3}I (http://www-spi.lip6.fr/~queinnec/VideoC/ps3i.html) entspricht ebenfalls lediglich R^{4}RS.

SIXX (http://dgym.homeunix.net/mediawiki/index.php?title=Sixx) lässt viele Bestandteile von R^{5}RS weg, da es als kleine Implementation zur Einbettung in Java-Applets konzipiert ist.

MSCHEME (http://mscheme.sourceforge.net/) lässt ebenfalls Teile des Standards weg und bietet keine Integration von Java-Objekten.

LLAVA (http://llava.org/) ist von der Konzeption her sehr interessant, da es vollständige Integration von Java und Scheme bieten soll. Leider ist LLAVA laut (http://llava.org/BUGS) (noch) nicht thread-safe und in der Dokumentation wird keine Aussage darüber gemacht, mit welchem Standard die Implementation konform ist. Das Hauptaugenmerk lag hier auf einem Lisp-ähnlichen Zugang zu Java und nicht auf einer standardkonformen Implementation von Scheme. Außerdem deutet u. a. das letzte Änderungsdatum von http://llava.org/ChangeLog darauf hin, dass diese Implementation seit 2005 nicht mehr aktiv weiterentwickelt wird.

BIGLOO (http://www-sop.inria.fr/mimosa/fp/Bigloo/) ist ein Compiler/Interpreter der Scheme nach C, Java oder C# übersetzen kann. Er soll vollständige Integration von Scheme und Java bieten. Es gibt aber recht viele Einschränkungen. Davon sind hier die wichtigsten:

- Die Prozedur *eval* akzeptiert nur eine Untermenge der Sprache.
- First-Class-Continuations (im C-Backend) sind so ineffizient, dass eine Compiler-Option benötigt wird, um sie anzuschalten. Im Java-Backend wird die Prozedur `call/cc` überhaupt nicht sinnvoll unterstützt.
- Es gibt nur eingeschränkte Unterstützung für Endrekursion.

KAWA (http://www.gnu.org/software/kawa/) Diese Implementation erfüllt fast alle Kriterien und bietet darüber hinaus eine Möglichkeit, die SISC nicht bietet: Java-Klassen können in Scheme erweitert werden (in SISC kann Scheme-Code bisher lediglich ein Java-Interface implementieren). Leider ist die Unterstützung für Endrekursion in KAWA so ineffizient, dass sie standardmäßig abgestellt ist, wodurch einige für Scheme typische Problemlösungen benachteiligt werden.

Auch sind die Continuations gegenüber dem Standard R^{5}RS stark eingeschränkt. Continuations können nicht erneut aufgerufen werden, nachdem sie bereits verlassen wurden. Bei SISC ist dies in wesentlich geringerem Maße der Fall, da Continuations dort lediglich über die Java-Scheme-Grenze hinweg eingeschränkt werden, nicht aber im Scheme-Code selbst. Dies ist ein entscheidender Nachteil von KAWA gegenüber SISC, da eine der beiden in Frage kommenden Implementierungen von deklarativer Programmierung mit Scheme (SCHELOG ([SCHELOG, 2003])) extensiv mit Continuations arbeitet. (Siehe Kapitel 9.1 (S. 113).)

Beim derzeitigen Stand der Implementation würde die Verwendung von KAWA in Scheme-Referenznetzen zu viele Einschränkungen bedeuten. Das Beispiel in Abbildung 8.15 (S. 109) ließe sich z. B. überhaupt nicht mit KAWA implementieren. Noch einschneidender ist die Tatsache, dass viele Beispiele aus Scheme-Lehrbüchern sich auf effiziente Endrekursion verlassen, was eine effiziente Unterstützung für Endrekursion bei der gewählten Scheme-Implementierung zwingend erscheinen lässt. Endrekursion als Synonym für Iteration ist ein Grundkonzept der Sprache Scheme.

Dennoch ist KAWA, vor allem da es noch aktiv weiterentwickelt wird, die beste Alternative zu SISC.

SISC (http://sisc-scheme.org/) Diese Implementation ist am besten geeignet, da sie R^{5}RS-Konform und thread-safe ist,

aktiv entwickelt wird und vollständige Java-Integration in beide Richtungen bietet. Einige relevante Details werden direkt im Folgenden erklärt, in Kapitel 10.1 (S. 123) folgen weitere Details zu dieser Implementation.

In den folgenden Unterabschnitten werden, parallel zur Argumentation im Abschnitt 12.1 von [Kummer, 2002], Kriterien für die Auswahl von Java als Anschriftsprache für RENEW mit Eigenschaften der Programmiersprache Scheme verglichen.

Da die Auswahl der Scheme-Implementation an dieser Stelle bereits getroffen wurde, werden die Kriterien aus [Kummer, 2002] nicht auf reines R^5RS-Scheme angewendet, sondern zum großen Teil bereits auf SISC.

Die Überschriften stehen für die entsprechenden Abschnitte in [Kummer, 2002], der Abschnitt „Mögliche Sprachen“ wurde bereits in Kapitel 4 (S. 29) behandelt und ist hier somit irrelevant und wird außer Acht gelassen.

4.4.2 Zwingende Kriterien

Anpassung an bestehende Programmteile

Im Unterschied zu der Situation, die Olaf Kummer beschreibt, habe ich mit RENEW bereits eine ausgereifte Umgebung zur Verfügung, die in Java implementiert ist. Aus diesem Grunde waren Java-Integration und „thread safety“ zwingende Kriterien für die Auswahl einer Scheme-Implementation, die gut mit RENEW zusammenarbeitet.

Implementierungssprache unterstützt Beschriftungssprache

Kummer argumentiert in [Kummer, 2002, S. 238], dass die Beschriftungssprache im gleichen Laufzeitsystem wie die Implementierungssprache laufen sollte. Dies ist bei den meisten genannten Scheme-Implementationen in Java der Fall. Zusätzlich wird mit der geforderten *vollständigen Integration* sichergestellt, dass auch das

Scheme-Plugin (und auch weitere Renew-Plugins, die darauf aufsetzen) selbst in Scheme entwickelt werden können.

Interessant ist Kummers Anmerkung, dass auf Grund der gleichen Laufzeitumgebung auf den gleichen Datentypen operiert werden müsse und darum nur leichte syntaktische Variationen zulässig sein dürften. Diese durch praktische Erwägungen begründete Forderung entsprach 2002 dem Stand der Technik, da es kaum nennenswerte Sprachimplementationen für die Java-VM gab. Mittlerweile sind jedoch einige Sprachimplementierungen auf Basis von Java erhältlich, die sich sowohl in den Datentypen als auch syntaktisch erheblich von Java unterscheiden und dabei die gewünschte vollständige Integration in beide Richtungen bieten (wie etwa JYTHON, JRUBY und auch SISC). Die technischen Schwierigkeiten, von denen Kummer spricht, werden durch diese bereits (zu großen Teilen) erfolgreich gelöst.

Das im Titel genannte Kriterium wird weiterhin dadurch erfüllt, dass die Implementation des Plugins selbst so weit wie möglich in Scheme erfolgt. Dadurch lassen sich die Beschriftungen leicht in der Implementationssprache bearbeiten und es wird der Anwenderin ermöglicht, die Implementation zu erweitern (siehe auch Abschnitt 4.4.2 rechts gegenüberliegend).

Objektorientiertheit

SISC bietet neben den Java-Klassen und -Objekten auch ein eigenes Objektsystem an, welches mit dem von Java harmoniert. Da Scheme multiple Paradigmen unterstützt, ist jedoch der objektorientierte Ansatz in den seltensten Fällen die Methode der Wahl, insbesondere wenn mit Funktionen höherer Ordnung und anderen typischen Vorzügen funktionaler Programmiersprachen komfortabler gearbeitet werden kann.

Aus diesem Grunde muss die Forderung erweitert werden: *Alle* Objekte erster Klasse in Scheme sollten auch als Marken in Referenznetzen verwendbar sein. Dies gilt also neben Netzinstanzen bzw. Netzexemplaren auch für Prozeduren und Umgebungen.

Continuations als Marken werden am Beispiel in Abschnitt 8.2.2 (S. 108) gezeigt.

Existenz eines Parsergenerators

Ein Parsergenerator ist für Scheme-Code, welcher in Scheme verarbeitet wird, vollkommen unnötig, da die grundlegende Datenstruktur der Sprache gleichzeitig die Form ist, in der Programmcode dargestellt wird. Der Operator `read` erzeugt aus Text, der Scheme-Ausdrücke beschreibt, Scheme-Objekte. Sofern es sich dabei um Listen handelt, können diese als S-Expressions möglicherweise legale Scheme-Programme beschreiben. Mit den Listenoperationen können also Programme wie jede andere Liste bearbeitet werden und dann mittels `eval` ausgeführt werden. Eine komfortablere Möglichkeit zur Einführung neuer syntaktischer und semantischer Konstrukte in die Sprache bietet aber das Makrosystem.

Nützlich bei der Analyse von Scheme-Anschriften ist die Möglichkeit Makros zu expandieren. SISC implementiert das in [R^{5}RS, 1998] spezifizierte hygienische Makrosystem auf Basis von `syntax-case` (siehe [Dybvig, 1992]). Mit der Prozedur `sc-expand` können Ausdrücke (in Form von S-Expressions) auf eine minimale Untermenge der Sprache zurückgeführt werden, die einfacher automatisch analysiert werden kann. Relevant wird dies unter anderem bei der Entscheidung, welche Bezeichner in einem beliebigen Scheme-Ausdruck unifiziert werden sollen. Kapitel 9 (S. 113) führt das Verfahren im einzelnen aus.

Möglichkeit des dynamischen Ladens von externen Klassen

SISC bietet hierbei sowohl die Möglichkeit, Java-Klassen mit Hilfe der üblichen Mechanismen nachzuladen, als auch Scheme-Programmtexte und -Module (mit Hilfe der `findResource`-Methode der Java-`ClassLoader`).

Das Nachladen von Java-Klassen in Java-Referenznetzen ist unter anderem notwendig, weil der interne Java-Parser in RENEW be-

wusst restriktiv gestaltet ist. In Deklarationsknoten sind, wie bereits erwähnt, nur Import- und Variablendeklarationen erlaubt, keine Klassendefinitionen.

Die Restriktion der Semantik von Scheme-Referenznetzen erfolgt aber über das in Kapitel 6 (S. 59) beschriebene Verfahren mit Hilfe von Umgebungen und nicht über eine rein syntaktische Restriktion des Parsers.

Darum kann die Syntax der in Deklarationsknoten erlaubten Scheme-Ausdrücke deutlich permissiver sein und es können unproblematisch einfache Programme in den Deklarationsknoten selbst beschrieben werden.

Die Darstellung einfacher und schon etwas komplexerer Scheme-Referenznetze kann darum in sich selbst abgeschlossen sein, während bei Java-Referenznetzen ab einem bestimmten Punkt die Semantik des Netzes nur in den importierten Klassendefinitionen nachgelesen werden kann. Dieser Punkt sollte bei Scheme-Referenznetzen prinzipiell erst dann eintreten, wenn die Definitionstexte zu umfangreich werden, um gut in der grafischen Darstellung des Referenznetzes Platz zu finden.

Automatische Freispeicherverwaltung

SISC verwendet die Freispeicherverwaltung von Java in einer Weise, dass Referenzen auf nicht mehr relevante Objekte einfach fallen gelassen werden. Die Verwaltung des dynamischen Speichers bleibt dann Java überlassen. In Sonderfällen nimmt SISC etwas mehr Einfluss auf die Freispeicherverwaltung, die entsprechenden Optimierungstechniken sind in [Miller, 2002] beschrieben.

Zukunftssicherheit

Der erste Artikel über Scheme wurde 1975 veröffentlicht [Suss:75, 1975], man kann also nicht sagen, dass Scheme sich im stürmischen Anfangsstadium seiner Entwicklung befindet. Es erscheint auch unwahrscheinlich, dass eine Sprache, die bereits über 30 Jahre lang

verwendet wird, plötzlich verschwindet, wie an anderen Beispielen wie Fortran oder COBOL leicht nachzuvollziehen ist.

Bedenken bezüglich der Zukunftssicherheit könnten sich aber durchaus auf die verwendete Implementation richten. SISC ist für eine Implementation einer Programmiersprache sehr jung, wenn man die Veröffentlichung von [Miller, 2002] im Jahre 2002 als Geburtsstunde ansieht.

Des Weiteren ist es fraglich, was nach der Veröffentlichung der sechsten Revision des Standards geschieht, da der am 12. August 2007 ratifizierte Entwurf sehr stark umstritten ist[8]. Man kann damit rechnen, dass einige Implementationen bei der alten Version des Standards bleiben, wie dies bereits beim Wechsel von R^{4}RS auf R^{5}RS aufgetreten ist (wie auch an der Liste der Java-basierten Scheme-Implementationen deutlich wird).

Das würde im schlimmsten Fall bedeuten, dass die mit RENEW ausgelieferte Scheme-Implementation nicht mehr dem allerneusten Stand der Technik entspricht. Auch wäre jede Scheme-Implementation, die den in 4.4.1 (S. 40) gegebenen Kriterien entspricht, geeignet für die Implementation der hier beschriebenen Konzepte, was zum Beispiel einen Nachfolger von SISC oder KAWA beinhalten könnte.

Es ist also interessant, den möglichen Portierungsaufwand auf eine andere, Java-basierte Scheme-Implementation zu beleuchten. Es zeigt sich hierbei, da sich alle Scheme-Implementationen naturgemäß in den Punkten voneinander unterscheiden, die *nicht* im Standard festgelegt sind, dass gerade die Integration von Java und Scheme ein entscheidender Punkt ist.

Andere mögliche Punkte sind das Modulsystem und wenige kleine Erweiterungen von SISC gegenüber R^{5}RS.

[8] Der Entwurf ist derzeit (September 2007) mit den formalen und informellen Kommentaren unter http://www.r6rs.org/ verfügbar. Anfängliche Bemühungen der R^{6}RS-Kritiker um eine Aktualisierung des fünften Standards ohne die Schwachpunkte des sechsten finden sich im Web unter http://scheme-punks.org/wiki/index.php?title=ERR5RS:Charter.

Relativ unkritisch hingegen dürften die auf dem SRFI-Prozess[9] beruhenden Erweiterungen sein, von denen SISC sehr viele unterstützt, da diese teilweise in portablem R5RS-Scheme geschrieben sind und außerdem von sehr vielen Scheme-Implementationen unterstützt werden.

Außerdem wird eine Einbettung von Logikprogrammierung in Scheme verwendet werden. Beide zur Diskussion stehenden Kandidaten sind aber in portablem R5RS-Scheme geschrieben.

Die Hauptaufgabe bei einer solchen Portierung bestünde also in der Anpassung der Java-Integration an eine hypothetische andere Scheme-Implementation in Java.

Unterstützung einer grafischen Bedienoberfläche

Dieses Kriterium ist für die Anschriftsprache weniger relevant und ergibt sich in [Kummer, 2002] eben daraus, dass die Beschriftungssprache identisch mit der Implementierungssprache der Anwendung ist.

SISC unterstützt auch durch den Zugriff auf Java alle in Java verfügbaren grafischen Oberflächen.

Behälterklassen

R5RS-Scheme ist schwach getypt und polymorph. Eine Liste kann Elemente verschiedenen Typs enthalten. Zur Typprüfung steht in den allermeisten Scheme-Implementationen und Dialekten nur dynamische Typprüfung zur Verfügung, was auch für SISC zutrifft. (Einige wenige Implementationen bieten optional statische Typprüfung an.)

Damit bietet Scheme von sich aus ein funktionales Äquivalent zu den (mittlerweile auch in Java direkt verfügbaren) Behälterklassen an.

[9] siehe http://srfi.schemers.org/

4.4.3 Weitere Kriterien

Portabilität

SISC ist so portabel wie Java. Es werden keine betriebssystem- oder plattformspezifischen Erweiterungen definiert, die über die Möglichkeiten der Java-Implementation hinaus gehen.

Da SISC aus der virtuellen Maschine heraus aufgerufen wird, müssen auch keine Besonderheiten des Betriebssystems bei der Prozessinteraktion berücksichtigt werden, wie dies bei einem externen Compiler oder Interpreter der Fall wäre.

Bei Scheme gilt es aber auch die Portabilität über verschiedene Scheme-Implementationen hinweg zu berücksichtigen. Der Unterabschnitt 4.4.2 (S. 46) enthält dazu bereits einige Anmerkungen und in Abschnitt 7.1.2 (S. 66) werden die Bedingungen für eine alternative Implementation als Einbettung in (R^5RS-)Scheme erörtert.

Effizienz

Momentan ist KAWA die einzige andere in Frage kommende, Java-basierte Scheme-Implementation, welche zusätzlich in Java-Bytecode übersetzt werden kann, was aber für Netzanschriften wenig Vorteile bringt, da diese schon in Java-Referenznetzen mittels der Reflection-API interpretiert werden. Es bieten sich also ohnehin wenig Vergleichsmöglichkeiten mit anderen Java-basierten Scheme-Implementationen.

Bei Bedarf kann bei SISC die Effizienz ein wenig gesteigert werden, indem die Scheme-Anschriften in ein Zwischenformat expandiert werden, welches die komplexen syntaktischen Bindungen in ihre primitiven Komponenten übersetzt.

Interessanter ist ein Effizienzvergleich mit Java-Anschriften, die in RENEW ebenfalls interpretiert werden, oder mit anderen Java-basierten Interpretern (vgl. Abschnitt 14.2.6 (S. 196)).

Abgesehen von der Sprachimplementation kann hier aber auch das Verfahren der Auswertung von Scheme-Beschriftungen problematisiert werden. Das in Kapitel 6 (S. 59) beschriebene Verfahren

verwendet die Scheme-Prozedur `eval` welche unter dem Gesichtspunkt der Effizienz die denkbar ungünstigste Wahl ist, da die Ausdrücke bei jeder Auswertung expandiert, optimiert, analysiert und schließlich compiliert werden müssen.

Dieses Verfahren hat jedoch den Vorteil, dass es sehr einfach zu spezifizieren und zu implementieren ist, was in Anbetracht der kurzen zur Verfügung stehenden Zeitspanne für diese Arbeit und die Implementation ein entscheidender Vorteil ist. Außerdem ist die sich daraus ergebende Semantik der Scheme-Referenznetze sowohl einfach als auch flexibel.

Wenn die erste Implementation korrekt funktioniert, bietet SISC einige Möglichkeiten, die Effizienz unter Beibehaltung der Semantik zu steigern (vgl. Abschnitt 15.2 (S. 201)).

Aber sicherlich werden Scheme-Referenznetze in RENEW auch bei sorgfältigster Implementation nicht ganz die Effizienz der Java-Referenznetze erreichen.

Auf keinen Fall kann bei der ersten Implementation im Rahmen dieser Arbeit optimale Performanz erreicht werden. Das Hauptgewicht muss bei dabei vielmehr auf der korrekten Umsetzung der Anforderungen und auf Verständlichkeit liegen.

Typsystem

Da Typprüfung in SISC und den meisten anderen Scheme-Implementationen ausschließlich dynamisch erfolgt, ergibt sich hieraus ein großer Unterschied zwischen der Implementation von Scheme-Anschriften und Java-Anschriften für Referenznetze. Insbesondere wurde mit großem Aufwand für RENEW die Möglichkeit implementiert, typfreie Anschriften zur verwenden, eben weil dies für die schnelle Entwicklung von Prototypen praktischer ist.

Dieser Aufwand ist bei Scheme unnötig.

Wenn Typprüfung gewünscht wird, ergibt sich ein etwas größerer Aufwand aus der Notwendigkeit, diese dynamisch und explizit mittels Prädikaten vorzunehmen.

Unterstützung von Nebenläufigkeit

Da der RENEW Simulator nebenläufig ist, heißt das, dass die Interpretation von Transitionsanschriften ebenfalls robust gegenüber Nebenläufigkeit sein muss, weswegen ich dies auch schon als Bedingung an eine Sprachimplementation formuliert habe.

Das SISC-Handbuch [Miller und Radestock, 2006] und die Dokumentation der Java-Klassen [SISC Javadoc] erwähnen explizit, welche Bestandteile der SISC-Laufzeitumgebung über Threads hinweg geteilt werden dürfen und welche für jeden Thread nur einmal existieren dürfen. Außerdem bietet SISC selbst Mechanismen zur Behandlung von Nebenläufigkeit und zur Synchronisation an.

Verfügbare Bibliotheken

Da durch die vollständige Integration von Java und Scheme alle Java-Bibliotheken auch in SISC zur Verfügung stehen und zusätzlich auch einige für Scheme ergibt sich hier ebenfalls kein Nachteil gegenüber der Sprache Java.

Ähnliches wird für jede andere Sprache gelten, die eine vollständige Integration mit Java bietet.

Preiswerte Entwicklungsumgebung

Sowohl für Java als auch für Scheme bieten sich diverse preiswerte oder kostenlose Entwicklungsumgebungen an. Da die Entwicklung von Scheme stärker interaktiv erfolgt als die von Java (einzelne Prozeduren und Ausdrücke können sofort an einer interaktiven Kommandozeile, der REPL[10] ausgewertet und getestet werden), ist es

[10] „Read-Eval-Print-Loop“ nach der Auswertungsreihenfolge der Operatoren in einem sehr kurzen Lisp-Programm, das eine rudimentäre Form dieser Kommandozeile zur Verfügung stellt. In Scheme:

```
(define (repl)
  (display (eval (read)))
  (newline)
  (repl))
```

nötig, dass für die Entwicklung der Scheme-Teile des Plugins eine solche REPL zur Verfügung steht.

Um diese Arbeitsweise zu unterstützen, wird als Teil des Scheme-Plugins eine solche REPL implementiert, die Zugriff auf eine laufende Instanz des Interpreters bietet. An dieser Kommandozeile können zunächst komplexere Anschriften (wie etwa rekursive Prozeduren) getestet werden, in einer späteren Ausbaustufe wäre es denkbar, einen interaktiven Debugger zur Verfügung zu stellen, mit dem sich z. B. die Umgebung einer schaltenden Transition untersuchen ließe.

Ein bedeutendes Problem bei der Programmierung in Lisp-Dialekten ergibt sich hingegen aus der Syntax dieser Sprachen. Da die Syntax vornehmlich für Maschinen (Compiler, Interpreter) einfach zu lesen ist, *muss* für den menschlichen Leser und Programmierer die Blockstruktur der Ausdrücke anhand von Einrückung verdeutlicht werden.

Damit bei fehlerhaftem Code die Diskrepanz zwischen der intendierten und der tatsächlichen Struktur des Programms (Klammerung) leichter deutlich wird, muss der Editor die korrekte Einrückung von Lisp-Ausdrücken *zwingend* unterstützen und sie halbautomatisch durchführen, wie dies z. B. die Editoren `emacs`, `edwin` oder auch `vi` (bei entsprechender Konfiguration) tun.

Auch das Zählen von schließenden Klammern ist eine völlig unzumutbare Tätigkeit sowohl beim Lesen als auch beim Schreiben von Scheme-Code, weswegen auch diese Tätigkeit vom Editor automatisiert oder unterstützt werden muss.

Auf lange Sicht wird es also nötig sein, diese Funktionalität auch im Scheme-Plugin für RENEW zu integrieren. Da vornehmlich einfache Ausdrücke als Beschriftungen verwendet werden, kann es dem Benutzer der frühen Versionen zugemutet werden, sich für komplexere Ausdrücke eines entsprechenden Editors zu bedienen und das korrekt eingerückte Ergebnis in die RENEW-Zeichnung zu übertragen.

In Scheme hat sich der Name der Funktion `print` gegenüber der entsprechenden Funktion in älteren Lisp-Dialekten (und heute noch in Common-Lisp) geändert.

Da bereits die Scheme-REPL in RENEW integriert wurde, liegt es ohnehin nahe, diese in einen der obengenannten Editoren zum Entwickeln und Testen komplexerer Scheme-Ausdrücke zu integrieren (wie etwa im „inferior scheme buffer“ im `emacs`).

4.4.4 Auswertung

Alle Kriterien wurden als gleichwertig zu denen in [Kummer, 2002] erkannt, jedoch wurde die Bewertung unter der Voraussetzung getroffen, dass bereits eine fertige Implementierung in Java zur Verfügung steht und lediglich eine weitere Anschriftsprache hinzugefügt wird.

Auf dieser Basis und mit den Ergebnissen aus Abschnitt 4.4.1 ist SISC die geeignetste Implementation.

Aus den vorigen Betrachtungen haben sich schon folgende konkrete Anforderungen an die Implementation des Plugins ergeben, die hier noch einmal kurz zusammengefasst werden:

- Scheme-typische Lösungen und Konventionen werden bevorzugt.

  Daraus folgt:

  - Die Entwicklung erfolgt weit wie möglich in Scheme.
  - Eine REPL wird zur Verfügung gestellt.
  - Der Komfort beim Editieren und der Fehlerbehandlung von Scheme-Beschriftungen sollte später erhöht werden.

- Alle Scheme-Objekte erster Klasse können auch Marken sein.

- Die Implementation muss robust gegenüber Nebenläufigkeit sein.

- Scheme wird zum Parsing der Beschriftungen verwendet.

- Es gibt keine syntaktische Restriktion der möglichen Anschriften, dafür wird der Sprachumfang mittels Umgebungen eingeschränkt.

- Das gesamte Plugin (inklusive aller SISC-Klassen) ist Bestandteil des RENEW-Klassenbaums. Es werden keine externen Programme aufgerufen, die erst vom Benutzer installiert werden müssen.

5 Syntax der Anschriften in Scheme-Referenznetzen

Scheme-Referenznetze können an jeder Stelle beliebige Scheme-Ausdrücke als Beschriftungen erhalten. Über die in der Standardumgebung enthaltenen Bindungen hinaus werden weitere Scheme-Operatoren eingeführt, damit mit Netzexemplaren gearbeitet werden kann und und zusätzliche Bindungen vom Benutzer eingeführt werden können. Diese Operatoren können in beliebigen Ausdrücken, also auch in benutzerdefinierten Prozeduren und Syntax verwendet werden.

Scheme-Operatoren alleine reichen aber zur Arbeit mit Referenznetzen nicht aus.

Es sollen zwar möglichst alle Beschriftungen in Scheme-Referenznetzen der üblichen Scheme-Syntax entsprechen, auch soll soweit wie möglich auf syntaktische Besonderheiten verzichtet werden, aber einige Anschriften verlangen dennoch eine Sonderbehandlung, weil sie eine Änderung im Verhalten einer Transition spezifizieren, die schon beim Übersetzen des Netzes berücksichtigt werden muss. Dies betrifft die Auszeichnung von manuellen Transitionen, sowie Uplink- und Downlink-Beschriftungen.

Außerdem sollten synchrone Kanäle nach dem in Abschnitt 1.3.4 (S. 13) aufgestellten und diskutierten Lokalitätsbegriff für Referenznetze eine besondere Auszeichnung erfahren. Die Lokalität der Transition soll ja direkt an der Beschriftung ablesbar sein. Wenn es aber möglich wäre, Uplink- oder Downlink-Beschriftungen wie normale Scheme-Operatoren in Makros oder Prozeduren zu verwenden, wäre diese Forderung verletzt. Damit ist für diese eine Sonderbehandlung *nicht nur* aus den im vorigen Absatz genannten prakti-

schen Gründen, sondern auch konzeptionell dringend erforderlich.

Auch wäre es nicht sinnvoll wenn die Auszeichnung einer manuellen Transition in beliebigen Scheme-Ausdrücken verwendet werden könnte, weil dann für den Benutzer nicht mehr erkennbar wäre, dass die Transition angeklickt werden muss, um schalten zu können.

Für die Erzeugung von Netzexemplaren sowie Action-Anschriften wird keine spezielle Syntax benötigt, da diese weder besonders ausgezeichnet werden müssen noch eine Sonderbehandlung beim Übersetzen des Netzes erfordern. Gleiches gilt für die beiden noch zu definierenden Bindungsoperatoren.

Da die genannten Spezialfälle nur an Transitionen auftreten können, muss lediglich die Syntax für Transitionsbeschriftungen genauer spezifiziert werden.

Die Syntax wird so gewählt, dass sie normaler Scheme-Syntax ähnelt, aber gleichzeitig deutlich erkennbar von ihr abweicht: Statt der runden Klammern werden für die Spezialfälle geschweifte Klammern verwendet.

Die formale Syntax von Transitionsanschriften in Scheme-Referenznetzen ergibt sich durch folgende Grammatik unter Verwendung der in [R^{5}RS, 1998, Abschnitt 7.1.3: Expressions] definierten Nonterminale `<expression>` und `<identifier>`:

```
<transition-inscription> ⟶ <transition-expression>*

<transition-expression> ⟶ <special-operation>
   | <expression>

<special-operation> ⟶ {manual}
   | {! <sisc-symbol> <expression>
        <expression>*}
   | {? <sisc-symbol> <expression>*}

<sisc-symbol> ⟶ <symbol>
   | <cased-symbol>

<cased-symbol> ⟶ |<identifier>|
```

Die Produktion für `<cased-symbol>` wurde [Miller und Radestock, 2006] entnommen. SISC unterscheidet Klein- und Großbuch-

staben bei in dieser Notation angegebenen Bezeichnern, so dass die meisten sinnvollen Namen für RENEW-Referenznetze, die auf ihren Dateinamen beruhen, als Parameter für `?` (Uplink-) und `!` (Downlink-Spezifikation) angegeben werden können.

Es wird bei `<special-operation>` von der üblichen Scheme-Syntax abgewichen, damit die Sonderstellung der hier eingeführten Operatoren deutlich wird und Verwechslungen mit Scheme ausgeschlossen werden. Da SISC in Abweichung von R^{5}RS die geschweiften Klammern als Teil des Bezeichners auffassen würde, kann diese Notation nicht in Scheme-Ausdrücken verwendet werden[1].

Es lassen sich aber durchaus Scheme-Bezeichner mit den Namen `manual`, `?`, oder auch `!` definieren und als Variablen oder Syntax verwenden. Es handelt sich bei diesen Namen *nicht* um reservierte Schlüsselwörter, sie werden lediglich in Verbindung mit der Notation in geschweiften Klammern an Transitionen gesondert interpretiert.

Bei der Übersetzung von Scheme-Referenznetzen werden also die Transitionsbeschriftungen zunächst auf die Produktion `<special-operation>` hin untersucht.

Die Operatoren `manual`, `?`, und `!` sind damit *keine* Scheme-Operatoren, sondern spezielle Auszeichnungen für Transitionsbeschriftungen. Die in `<special-operation>` definierten Operatoren werden nicht als Ausdrücke aufgefasst, hingegen werden ihre `<expression>`-Parameter als Scheme-Ausdrücke interpretiert. Für diese sind auch weiterhin runde Klammern zu verwenden

Die Syntax ist der S-Expression-Syntax weiterhin ähnlich, so dass der Scheme-Parser mit Hilfe einer kleinen Anpassung die speziellen Auszeichnungen leicht parsen kann.

[1] Wenn der in [Miller und Radestock, 2006] beschriebene strikte R^{5}RS-Modus verwendet wird, sind die geschweiften Klammern einfach illegale Zeichen und können ebenfalls nicht in Scheme-Ausdrücken verwendet werden. Der Standard gibt aber nicht vor, wie die reservierten Zeichen `{}[]` behandelt werden müssen. PLT Scheme z. B. erlaubt die Verwendung von geschweiften Klammern als Alternative zu runden Klammern.

Diese Art der Sonderbehandlung ist aber für keine andere Beschriftung nötig. Action-Anschriften und spezielle Bindungsoperatoren können beliebig in Scheme-Ausdrücken verwendet werden.

Prinzipiell gilt das gleiche für Netzexemplare. Es werden aber keine Bindungen bereitgestellt, mit denen Uplinks oder Downlinks in Scheme-Ausdrücken aufgerufen werden können. Da der Zustand des Simulators in den Scheme-Anschriften nicht sichtbar gemacht wird, ist es für den Anwender schwierig diese Bindungen selbst nachzurüsten, aber auch nicht unmöglich, so lange das von SISC bereitgestellte Java-Interface in Beschriftungen verwendet werden kann. Am Ende des nächsten Kapitels findet sich aber eine Möglichkeit, wie solche Anomalien verhindert werden können.

Wie bei Java-Referenznetzen soll die Syntaxprüfung bereits beim Erstellen des Netzes in der grafischen Oberfläche erfolgen.

Interessant ist der Umstand, dass durch die hier festgelegte Syntax alle Transitionen, deren Aktivierung nicht direkt an den Stellen im Vorbereich ablesbar ist, eine `<special-operation>`-Auszeichnung erhalten. Zur Bestimmung der Lokalität muss also in der grafischen Darstellung eines Scheme-Referenznetzes nur auf die Kanten und die speziellen Operatoren geachtet werden, alle anderen Beschriftungen an Kanten oder Transitionen sind für den in Definition 4 (S. 15) vorgeschlagenen Lokalitätsbegriff irrelevant.

6 Auswertung von Ausdrücken in Scheme-Referenznetzen

Wie bereits erwähnt soll die Auswertung von Scheme-Ausdrücken mit der Scheme-Prozedur `eval` erfolgen, die als zweiten Parameter eine Auswertungsumgebung erhält, in der alle Bindungen definiert sein müssen, die im auszuwertenden Ausdruck verwendet werden sollen.

Die ausgewählte Scheme-Implementation SISC enthält neben den in [R^{5}RS, 1998] definierten Prozeduren `eval`, mit der auch Bindungen in der übergebenen Umgebung verändert werden können, und den Prozeduren `scheme-report-environment`, `null-environment` und `interaction-environment` zum Erzeugen vordefinierter Auswertungsumgebungen *zusätzlich* noch die Prozeduren `make-child-environment` und `parent-environment` mit denen Umgebungen miteinander verkettet werden können, die Prozedur `sisc-initial-environment`, mit der die Standardumgebung von SISC ohne benutzerdefinierte Bindungen erhalten werden kann und die Prozeduren `getprop` und `putprop`, mit denen, ohne die Verwendung von `eval` oder `set!`, eine Umgebung von außen verändert werden kann.

Darüber hinaus erlaubt SISC auch Definitionen im ersten Argument zu `eval`, was von R^{5}RS nicht spezifiziert wird, aber praktisch ist, um syntaktische Schlüsselwörter in der Umgebung einzuführen oder zu modifizieren.

In den verketteten Umgebungen wird dafür gesorgt, dass die Kind-Umgebungen keine Bindungen in der Eltern-Umgebung verändern können, was ideal für das Vorhaben ist, globale Bindungen vor Nebeneffekten zu schützen.

globale Umgebung

Übergangsfunktion

sekundäre Umgebungen

primäre Umgebung

Es liegt darum nahe, im Deklarationsknoten eines Scheme-Referenznetzes eine **globale Umgebung** für das *Netzmuster* zu definieren und außerdem eine **Übergangsfunktion**[1], die bei *jeder* Auswertung einer Transitionen-, Stellen- oder Kanteninschrift aus der globalen Umgebung eine *neue* Umgebung erzeugt, die bei der Auswertung verwendet wird. Diese Umgebungen werden ab jetzt als **sekundäre Umgebungen** bezeichnet. Die globale Umgebung kann demzufolge auch als **primäre Umgebung** betrachtet werden.

Die Übergangsprozedur muss deterministisch sein, in dem Sinne, dass die bei jedem Aufruf der Prozedur erzeugten Umgebungen jeweils die gleichen Objekte (Syntax oder Speicherplatz) binden müssen und keine Bindungen hinzukommen oder fehlen dürfen. Sie muss für die spezifizierte globale Umgebung eine Umgebung als Rückgabewert liefern können und außerdem darf sie selbst keine Nebeneffekte in der Umgebung erzeugen.

In einem frühen Entwurf sollten beliebige Prozeduren die Übergangsfunktion implementieren können, dies wäre aber aus mehreren Gründen sehr problematisch:

- Die Syntaxprüfung soll bereits beim Beschriften des Netzes erfolgen können. Damit aber nicht nur die S-Expression-Syntax mit der Prozedur `read` sondern auch die richtige Verwendung syntaktischer Konstrukte wie `lambda` oder `define` überprüft werden kann, müssen die Ausdrücke bereits beim Beschriften des Netzes probeweise expandiert werden.

  Dies muss aber schon in einer geeigneten sekundären Umgebung geschehen, insbesondere damit die hier zur Verwendung in Referenznetzen beschriebenen Operatoren ebenfalls überprüft werden können.

  Wenn aber beliebige Übergangsprozeduren erlaubt wären, könnten diese z. B. auch syntaktische Bindungen gegenüber

[1] implementiert durch eine Übergangsprozedur; ich rede aber im Folgenden vereinfachend von „der Übergangsfunktion“ auch wenn eine Prozedur gemeint ist, die die Übergangsfunktion implementiert

der primären Umgebung verändern und die Syntaxprüfung wäre dann nicht mehr verlässlich.

- Beliebige Übergangsprozeduren könnten z. B. Bindungen in der Umgebung bei jedem Aufruf abwandeln, so dass im Extremfall das Verhalten eines Netzes gar nicht mehr aus seinen Beschriftungen erkennbar wäre. Da bereits beliebige Scheme-Ausdrücke als Beschriftungen erlaubt sind, soll aber wenigstens die sekundäre Umgebung eine einfache Semantik haben.

- Zur Optimierung des Laufzeitverhaltens im Simulator ist es vorteilhaft, wenn die Beschriftungen bereits beim Übersetzen des Netzes so weit wie möglich analysiert und in ausführbare Form gebracht werden können. Dies ist nur dann möglich, wenn sich die Bindungen in der sekundären Umgebung nicht gegenüber der primären verändern.

Aus diesen Gründen sollte der Benutzer die Übergangsprozedur nicht frei spezifizieren können, sondern lediglich die Auswahl zwischen wenigen sinnvollen Werten erhalten.

Denkbare Werte für die globale Umgebung wären nun konkret die Rückgabewerte der SISC-Prozeduren `interaction-environment` und `sisc-initial-environment` (dieser ist auch eine sinnvolle erste Näherung für die Voreinstellung) oder auch des Aufrufs `(scheme-report-environment 5)`.

Eine sinnvolle Voreinstellung für die Übergangsfunktion ist `make-child-environment`, da auf diese Weise die Bindungen in der globalen Umgebung vor Veränderungen durch Transitionenbeschriftungen geschützt werden. Ein sinnvoller Wert wäre auch eine Implementation der Identitätsfunktion – typischerweise implementiert durch eine Prozedur

```
(lambda (x) x)
```

– wenn genau solche Nebeneffekte gewünscht sind. Dies scheinen nach den obigen Kriterien momentan die einzig sinnvollen Kandidaten zu sein.

Für den Benutzer des Plugins muss das System der Übergangsfunktionen also gar nicht erklärt werden. Mit den beiden möglichen Werten geht es aus Benutzersicht lediglich darum, ob die Bindungen in der primären Umgebung vor Mutation geschützt werden oder nicht.

Der Deklarationsknoten in einem Scheme-Referenznetz soll einen Wert eines benutzerdefinierten Typs zurückgeben, der aus einer Umgebung (was mit dem SISC-spezifischen Prädikat `environment?` getestet werden kann) und einer Prozedur besteht, welche als Argument eine Umgebung erhält und eine Umgebung als Rückgabewert zurück gibt. Da nur bestimmte Übergangsprozeduren erlaubt sind, können diese ebenfalls bei der Konstruktion des Rückgabewerts überprüft werden.

Die Beschriftung des Deklarationsknotens selbst kann nicht in der Voreinstellung für die globale Umgebung ausgewertet werden, sofern diese direkt von `(sisc-initial-environment)` abgeleitet wurde, da noch zusätzliche Bindungen zur Verfügung gestellt werden sollen, welche die Spezifikation von globaler Umgebung und Übergangsfunktion komfortabler machen sollen.

Der benutzerdefinierte Typ kann ein „Record“ nach [SRFI-9, 1999] oder eine Erweiterung davon sein, wie z. B. eine SISC-Klasse; ein einfaches Paar könnte auch ausreichen, da bei der Übersetzung des Netzes ohnehin die Elemente explizit überprüft werden müssen. Benutzerdefinierte Records oder Klassen bieten den Vorteil, dass sie sich etwas besser kapseln lassen. Eine solche Kapselung kann aber leicht umgangen werden, wenn Records in der verwendeten Scheme-Implementation durch einen primitiven Datentyp (wie etwa Vektoren oder Listen) implementiert werden, oder wenn eine Klasse verwendet wird und die Implementation einen Reflektionsmechanismus für Objekte bereit stellt. (Was bei SISC der Fall ist.)

Wenn der Deklarationsknoten leer ist, werden die globale Umgebung und die Übergangsprozedur auf einen Standardwert gesetzt. Die erste Näherung sei

```
(make-child-environment (sisc-initial-environment))
```

für die globale Umgebung und

```
make-child-environment
```

für die Übergangsprozedur. Damit werden Eingriffe in die Umgebung des Scheme-Plugins selbst und Eingriffe aus einer sekundären in die globale Umgebung ausgeschlossen.

Um Abweichungen von der Standardeinstellung komfortabel zu ermöglichen, wird die Implementation weitere Operatoren in der Form von vordefinierten Prozeduren oder Makros zur Verfügung stellen.

Daraus folgt, dass der obige Vorschlag für die Voreinstellungen für primäre Umgebung und Übergangsfunktion noch ein wenig zu einfach ist. Es müssen zum `(sisc-initial-environment)` mindestens noch die Bindungen hinzutreten, die zum Arbeiten mit Netzinstanzen, Actions, Gleichheitsspezifikationen usw. benötigt werden. Wie die Voreinstellungen für primäre Umgebung und Übergangsfunktion im einzelnen in der Praxis aussehen, wird im Teil „Umsetzung“ (S.123) genauer untersucht.

Die Auswertung von Beschriftungen geht dann nach folgendem Schema vor:

1. Bei der Übersetzung des Netzes wird zunächst der Deklarationsknoten nach dem oben erklärten Schema überprüft und ausgewertet, um die globale Umgebung und die Übergangsprozedur zu erhalten. Das Verfahren wird in Abschnitt 7.3 (S. 70) näher ausgeführt.

2. Dann werden die Anfangsmarkierungen der Stellen jeweils in einer neu erzeugten *sekundären* Umgebung ausgewertet.

3. Transitionen schalten nach dem in Kapitel 9 (S. 113) erklärten Verfahren unter Berücksichtigung der in Abschnitt 7.5 (S. 72) beschriebenen Typen von Transitionenbeschriftungen ebenfalls in einer frisch erzeugten *sekundären* Umgebung.

   Dabei könnte *konzeptionell* der gesamte Schaltvorgang einer Transition in eine Prozedur gefasst werden, die als Parameter

die Belegungen der Ausdrücke an den Eingangskanten erhält und die Belegungen der Ausdrücke an den Ausgangskanten zurück gibt. Eine solche Umsetzung wäre für eine Implementation in Scheme denkbar und wird in 9.3 (S. 116) genauer untersucht.

Praktisch wird sich aus der gewünschten engen Integration mit dem RENEW Simulator ein etwas anderes Vorgehen ergeben.

Pro Bindungssuche bzw. pro Schaltvorgang ist nur eine sekundäre Umgebung nötig.

(Nebeneffekte in der sekundären Umgebung können also bei mehrfacher Auswertung von Beschriftungen durchaus mehrfach und in zufälliger Reihenfolge auftreten.)

Wenn auf Nebeneffekte der Beschriftungssprache verzichtet werden soll und kann, müsste sich nun (mit etwas Aufwand) eine Umgebung definieren lassen, in der lediglich eine nebeneffektfreie Untermenge des R^{5}RS-Standards verfügbar ist (aber siehe Abschnitt 14.2.1 (S. 188)).

Als Erweiterung dieses Konzepts bietet es sich an, ein Scheme-Modul zur Verfügung zu stellen, mit dem sich leicht aus einer prototypischen globalen Umgebung nebst Übergangsprozedur ein neuer Formalismus bilden lässt, so dass nicht für jedes Netz die Umgebung explizit spezifiziert werden muss (oder kann) und außerdem für jedes Netz, das einen so gebildeten Formalismus verwendet, die gleichen Zusicherungen gelten.

Damit könnte z. B. die Anforderung erfüllt werden, dass alle Netze die nach dem gleichen Formalismus simuliert werden, Nebeneffekte in Beschriftungen, oder die Verwendung von Java-Objekten und Methoden verbieten. Mit dem zweiten Beispiel ließe sich auch verhindern, dass Beschriftungen Java-Methoden verwenden um den Zustand des Simulators abzufragen oder zu beeinflussen.

Nachdem nun das Prinzip der Auswertung von Netzanschriften grundsätzlich geklärt ist, erklärt das nächste Kapitel die Besonderheiten der einzelnen Beschriftungsarten.

7 Arten von Scheme-Netzanschriften und Netzelementen

Bevor die Anschriften und Netzelemente im einzelnen beschrieben werden, erfolgt eine grobe Einordnung der Beschriftungen in zwei konzeptionelle Stufen und grundsätzliche Überlegungen zu einer alternativen Implementation.

Scheme-Referenznetze sollen die gleiche Ausdrucksmächtigkeit wie Java-Referenznetze bieten, also muss in den folgenden Abschnitten die Syntax und Semantik aller in Java-Referenznetzen möglichen Beschriftungen sinnvoll auf Scheme übertragen werden. Außerdem bieten RENEW-Netze eine Vielzahl von Kantentypen, deren Bedeutung ebenfalls angepasst werden muss.

Vor der Erläuterung der Anschriftstypen muss erwähnt werden, dass Netzexemplare in Scheme-Referenznetzen Objekte wie alle anderen sein sollen. Sie können anders als bei Java-Referenznetzen nicht nur an Transitionen, sondern jederzeit erzeugt werden. Näheres dazu erklärt der Abschnitt 7.5.3 (S. 83).

7.1 Gefärbte Netze mit Scheme und Scheme-Referenznetze

7.1.1 Konzeptionelle Stufen

Zusätzlich zu der in Kapitel 5 (S. 55) erklärten syntaktischen Zweiteilung in spezielle Operatoren für Transitionsbeschriftungen und normale Scheme-Ausdrücke lässt sich noch eine Einteilung in kon-

zeptionelle Stufen vornehmen, die sich anhand der Anschriftstypen klar unterscheiden lassen.

Scheme-Coloured-Nets

Den eigentlichen Scheme-Referenznetzen lässt sich das Konzept der mit Scheme gefärbten Netze (bzw. **Scheme-Coloured-Nets**, in Anlehnung an [Reinke, 2000]) vorlagern. Diese haben beliebige (durch Scheme-Ausdrücke angegebene) Scheme-Objekte als Marken und beliebige Scheme-Ausdrücke als Beschriftungen. Dazu können noch sämtliche in RENEW möglichen Kantentypen, die im Folgenden erklärten Unifikations- und Bindungsoperatoren, sowie Action-Beschriftungen und die `{manual}`-Auszeichnung hinzutreten, ohne dass eine qualitativ neue Stufe erreicht wird.

Scheme-Referenznetze

Scheme-Referenznetze erweitern die Scheme-Coloured-Nets um Netzexemplare und synchrone Kanäle.

Als Beschriftungen kommen in dieser Stufe ein Operator zur Erzeugung von Netzexemplaren, eine Prozedur die das aktuelle Netzexemplar zurückliefert und die Auszeichnungen von Uplinks und Downlinks hinzu.

7.1.2 Alternative Implementationen

Beide konzeptionelle Stufen könnten alternativ auch als eigenständiger Formalismus in Scheme eingebettet werden, so dass durch einen Satz geeigneter Prozeduren und Makros mit den mächtigen Synchronisationsmechanismen gefärbter Netze bzw. der Referenznetze in Scheme-Anwendungen gearbeitet werden könnte. Es wäre dafür sinnvoll, die komplette Semantik der Netze in Scheme zu implementieren, da die Anbindung eines in Java geschriebenen Simulators an eine reine Scheme-Anwendung kaum vertretbar wäre.

Es wäre aus Portabilitätsgründen sehr wünschenswert wenn die gesamte Einbettung in reinem R^5RS erfolgen könnte, so dass die Benutzerin der Bibliothek die für ihre Zwecke geeignetste Scheme-Implementation selbst auswählen kann.

Aus diesem Grund habe ich versucht bei der Spezifikation der Syntax und des Verhaltens möglichst wenig Zugeständnisse an Java bzw. RENEW als Plattform zu machen. Die wenigen nötigen syntak-

tischen Abwandlungen in Kapitel 5 (S. 55) müssten aber vermutlich für eine Einbettung in Scheme anders gelöst werden. Für diese sollte aber ohnehin noch ein Format spezifiziert werden, in dem Referenznetze als Scheme-Programme beschrieben werden können.

Es wird auch eine Prozedur benötigt, die ähnlich wie `sisc-initial-environment` eine Anfangsumgebung mit einem Mindestumfang an benötigten Bindungen zur Verfügung steht. Wenn nur R^{5}RS-Funktionalität benötigt wird, wäre dies `(lambda ()(scheme-report-environment 5))`. Etwas schwerer fällt schon Implementation einer Prozedur, die ähnlich wie `make-child-environment` geschützte verkettete Umgebungen implementiert. Manche Scheme-Implementation bieten die Möglichkeit, als zweites Argument für den `eval`-Operator ein Modul zu verwenden, wodurch sich u. U. die hier beschriebene Semantik mit Hilfe eines geeigneten Modulsystems implementieren lässt.

Auch das erste Argument von `eval` ist nicht unproblematisch. Erweiternd zu R^{5}RS erlaubt SISC Definitionen in den ausgewerteten Ausdrücken, was sich bequem dazu nutzen lässt, Syntax wie z. B. `is?` oder `action` in die Auswertungsumgebungen der Netzbeschriftungen einzufügen. In einem Scheme in dem z. B. ein Modul als zweites Argument des `eval`-Operators verwendet werden kann, wäre dies kein Problem, weil die benötigten Bindungen in dem entsprechenden Modul definiert werden könnten. In einer Scheme-Implementation die weder Definitionen im ersten Argument, noch Module als zweites Argument zu `eval` erlaubt, noch eine andere Erweiterung zum Arbeiten mit Umgebungen gegenüber dem Standard bietet, müsste vermutlich ein metazirkulärer Evaluator (vgl. z. B. [Abelson u. a., 1985]) verwendet werden, um die richtige Semantik sicherzustellen. Da dies vermutlich nicht sehr effizient wäre, ist die Implementation als Einbettung in *reinem* R^{5}RS-Scheme leider keine einfache Aufgabe. Aber auch eine Implementation von Scheme-Referenznetzen, in der die Anforderungen der geschützten sekundären Umgebungen und der Nebeneffektfreiheit aufgegeben werden, wäre noch nützlich.

Die in dieser Arbeit beschriebene Implementation als RENEW-

Plugin mit Hilfe der Scheme-Implementation SISC ist also nur eine mögliche praktische Umsetzung von Scheme-Coloured-Nets und Scheme-Referenznetzen, die sich aber aus den bereits beschriebenen Gründen am einfachsten realisieren lässt. RENEW bietet außerdem als entscheidende Vorteile bereits einen fertigen Simulator sowie eine grafische Oberfläche an. Es besteht auch die Möglichkeit, dass Scheme-Referenznetze über synchrone Kanäle mit Java-Referenznetzen oder anderen in RENEW implementierten Netzformalismen kommunizieren können.

Aber eine Einbettung in Scheme bleibt auch dann noch sehr attraktiv, wenn eine fertige Implementation des Scheme-Plugins für RENEW bereits vorliegt, da dann Netze aus dem grafischen in das Scheme-Format und wieder zurück konvertiert werden könnten, ähnlich wie dies für MARIA in meiner Studienarbeit [Delgado Friedrichs, 2006] bereits geschehen ist.

Auch kann eine Einbettung in Scheme aus Performanzgründen interessant sein, da es Scheme-Implementationen gibt, die Scheme in effizienten C-Code übersetzen. Damit wäre das Scheme-Plugin für RENEW als Prototyping-Tool und grafische Entwicklungsumgebung für nebenläufige Scheme-Anwendungen geeignet, sofern die Scheme-Einbettung in der gewünschten Scheme-Implementation lauffähig ist. Auch aus diesem Grund wäre es sehr wünschenswert, die Implementation einer Scheme-Einbettung in möglichst reinem R^5RS-Scheme zu halten.

7.2 Eigenschaften aller Beschriftungen

Die folgenden Eigenschaften gelten für alle Beschriftungen an allen Netzelementen.

Es ist es ein *Fehler*, wenn ein Ausdruck in einer Beschriftung mehrere Werte (etwa mit der Prozedur `values`) zurück gibt. Dies gilt nur für den Rückgabewert des gesamten Ausdrucks, eine Verwendung z. B. von `values` innerhalb eines Aufrufs des Operators `call-by-values` ist natürlich legal.

Wenn der Benutzer mehrere Ausdrücke in einer Beschriftung einer Kante, Stelle oder Transition angibt, wird jeder Ausdruck als einzelne Beschriftung (also als einzelne Marke, Guard, usw.) interpretiert. Die textuelle Reihenfolge der Ausdrücke, auch in einer einzelnen Beschriftung, hat *keinen* Einfluss auf die Reihenfolge ihrer Auswertung. Mehrere Ausdrücke in einer Beschriftung werden generell *nicht* implizit sequenziert.

Sofern eine Berechnung eines Ausdrucks mittels einer Befehlssequenz intendiert ist, kann diese explizit mittels `begin` oder anderen Konstrukten erfolgen.

Im Deklarationsknoten dagegen werden die Anschriften sequenziert und der Rückgabewert der Sequenz bestimmt die globale Umgebung und die Übergangsprozedur.

7.2.1 Erläuterung zu den Operatoren

Alle im Folgenden eingeführten Operatoren werden analog zur Praxis in R^5RS als *syntax* oder *procedure* gekennzeichnet. Die besonderen Auszeichnungen aus Kapitel 5 (S. 55) werden mit *special* gekennzeichnet.

Syntaktische Variablen werden kursiv gesetzt. Die Art des Rückgabewertes wird mit

$\Longrightarrow$ *syntaktische-variable*

angedeutet.

Bei den Operatoren für die Verwendung in Scheme-Referenznetzen, wurde so weit wie möglich auf spezielle Notation verzichtet, d.h. sowohl Syntax als auch Prozeduren lassen sich wie Prozeduraufrufe an beliebiger Stelle der Beschriftung verwenden. Die besonderen Auszeichnungen aus Kapitel 5 bilden eine Ausnahme, aber ihre Notation ist der Notation von Scheme-Prozeduraufrufen ebenfalls sehr ähnlich. Diese Entscheidung steht im Einklang mit dem generellen Verzicht auf syntaktische Besonderheiten, der bei Scheme noch stärker als bei anderen Lisp-Dialekten zu Tage tritt.

Die zurückgegebenen Werte aller anderen Ausdrücke stehen (bei

erfolgreicher Unifikation) für die Fortsetzung der Berechnung zur Verfügung.

7.3 Deklarationen

Der Rückgabewert des Deklarationsknotens ist das bereits in Kapitel 6 (S. 59) beschriebene Aggregat aus der globalen Umgebung und der Übergangsprozedur. In Java-Referenznetzen können an dieser Stelle Import- und Variablendeklarationen stehen, also ist es sinnvoll, wenn bei Scheme-Referenznetzen ebenfalls Module und Bibliotheken importiert werden können.

Im Gegensatz zu Java-Referenznetzen soll auf syntaktische Beschränkungen so weit wie möglich verzichtet werden, also sind in Deklarationen alle möglichen Scheme-Programme erlaubt.

7.3.1 Auswertung des Deklarationsknotens

Zusammen mit der Festlegung, dass der Deklarationsknoten eine Umgebung und eine Prozedur liefern soll, ergibt sich folgendes Schema der Auswertung eines Deklarationsknotens:

1. Die Deklarationen werden zunächst in der Voreinstellung für die globale Umgebung ausgewertet (siehe Kapitel 6). Sowohl der Rückgabewert des Ausdrucks als auch die Umgebung selbst wird gespeichert.

2. Entspricht der Rückgabewert dem in Kapitel 6 festgelegten Typ, dann hat der Benutzer explizit eine Umgebung und eine Übergangsfunktion spezifiziert und diese werden entsprechend verwendet. Die gespeicherte Umgebung wird in diesem Falle nicht mehr benötigt.

3. Sofern der Rückgabewert *nicht* den entsprechenden Typ hat, wird statt dessen die gespeicherte Umgebung als globale Umgebung verwendet und die Übergangsprozedur erhält den Standardwert.

> Mit dieser Technik lassen sich auch ohne Kenntnis der genauen Semantik von Scheme-Referenznetzen ganz naiv Prozeduren und andere Bezeichner im Deklarationsknoten deklarieren, die in anderen Beschriftungen verwendet, aber ohne weitergehende Sachkenntnis nicht verändert werden können.

Zu der gewählten Umgebung müssen noch einige der im Folgenden erklärten Beschriftungen (`new`, `bind`, `protected-environment`, usw.) als Bindungen hinzutreten. Diese werden, sofern es sich um Variablenbindungen handelt, mit der Prozedur `putprop` eingefügt, im Fall syntaktischer Bindungen werden diese mit *`eval`* in die Umgebung eingefügt.

7.3.2 Spezifikation von Umgebung und Übergangsprozedur

```
procedure:  (protected-environment environment)
               ⟹ environment+transformer
procedure:  (unprotected-environment environment)
               ⟹ environment+transformer
```

Diese beiden Prozeduren stehen im Deklarationsknoten zur Verfügung um die primäre Umgebung und die Übergangsprozedur für das Netzexemplar zu bestimmen. Sofern `protected-environment` oder `unprotected-environment` den Rückgabewert des Deklarationsknotens liefern, wird danach die globale Umgebung und die Übergangsprozedur für das Netz bestimmt.

Im Falle von `protected-environment` wird `make-child-environment`, andernfalls eine spezifische Identitätsprozedur *`identity`* als Übergangsprozedur verwendet. Lediglich diese beiden Werte sind legale Werte für die Übergangsprozedur. Äquivalente Implementationen dieser beiden Funktionen werden nicht als legale Werte akzeptiert, da Prozeduren in SISC (und einigen anderen Implementationen) nur sinnvoll mit dem Prädikat `eq?` verglichen werden können, das nur vergleicht ob zwei Objekte die gleiche Referenz haben.

Sie können zwar an beliebiger Stelle verwendet werden, aber das Rechnen mit ihrem Rückgabewert wird dadurch erschwert, dass die dazu nötigen Operatoren nicht in der Umgebung sichtbar sind. Sofern einer der beiden Operatoren mit `(interaction-environment)` verwendet wird, werden die Operatoren `protected-environment` und `unprotected-environment` selbst zwangsläufig auch in den sekundären Umgebungen sichtbar.

7.4 Initiale Markierungen von Stellen

Stellen sind in Scheme-Referenznetzen grundsätzlich ungetypt, so wie auch alle Scheme-Variablen. Typprüfung kann an Transitionen mit speziellen Prädikaten erfolgen.

Initiale Markierungen werden in einer sekundären Umgebung ausgewertet. Dabei kann wie bei Java-Referenznetzen jede Stelle mehrere Inschriften besitzen, diese werden dann als Multimenge aufgefasst.

Jeder einzelne Ausdruck einer Beschriftung spezifiziert dabei *genau ein* Objekt der initialen Markierung einer Stelle. Das heißt z. B.: Eine Scheme-Liste ist *ein* solches Objekt und wird bei der Unifizierung auch nur als genau ein Objekt berücksichtigt.

Listen sollen aber mit speziellen Kantentypen aus Markierungen (also Multimengen) erstellt, bzw. in Markierungen aufgelöst werden können. Siehe Abschnitt 7.6 (S. 90).

7.5 Transitions- und Kantenanschriften

Alle Scheme-Ausdrücke, die an Transitionen verwendet werden können, können ebenfalls an Kanten verwendet werden. Die speziellen Auszeichnungen können nicht an Kanten verwendet werden.

Transitionsanschriften zerfallen bei Java-Referenznetzen in solche, die für die Unifikation (siehe Kapitel 9.1 (S. 113)) eine Rolle spielen, was auch für alle Kanten- und Stellenbeschriftungen gilt, und in solche, die einen Nebeneffekt erzeugen und darum explizit erst nach der Unifikation ausgewertet werden. (Außerdem wer-

den im RENEW-Simulator alle Kanten und Kantenbeschriftungen als Transitionsbeschriftungen aufgefasst.)

Für die Auswertung von Scheme-Beschriftungen wurde bereits auf syntaktische Einschränkungen verzichtet und statt dessen ein semantischer Mechanismus eingeführt. Es liegt darum nahe, für die verschiedenen Beschriftungstypen keine besondere Notation einzuführen, sondern sie wie (syntaktische oder prozedurale) Operatoren der Sprache Scheme zu behandeln. Diese Operatoren werden mit dem gleichen Mechanismus bereitgestellt der bereits zur Kapselung der Transitionsanschriften eingeführt wurde: Sie werden (aus Effizienzgründen) bereits in die primäre Umgebung eingefügt, können zum großen Teil aber erst in den sekundären Umgebungen verwendet werden.

Wie bereits in Kapitel 5 (S. 55) diskutiert müssen einige Beschriftungen aber einen Sonderstatus erhalten.

Wie bei Java-Beschriftungen werden Nebeneffekte in keiner Beschriftung explizit verboten. Sollen Nebeneffekte in einem Netz unterbunden werden, muss eine entsprechende primäre Umgebung mit passender Übergangsfunktion definiert werden.

Gerade weil Nebeneffekte in Scheme-Referenznetzen nicht prinzipiell verboten sind, muss wie bei Java-Referenznetzen ein Mechanismus angeboten werden, mit dem diese gesondert ausgezeichnet und behandelt werden können.

Dafür wird ein Makro `action` bereitgestellt, was zu einem ähnlichen Verhalten führt, wie das Schlüsselwort `action` in Java-Referenznetzen.

Außerdem wird ein Äquivalent zum Schlüsselwort `manual` bereitgestellt, welches festlegt, dass eine Transition nur manuell ausgelöst werden kann.

Alle Anschriften an einer Transition werden zusammen mit den Kantenbeschriftungen unifiziert.

7.5.1 Unifizierbare Transitionsanschriften

Grundsätzlich sind alle Transitionsanschriften außer der Spezifikation `{manual}` unifizierbar. Obwohl die Auszeichnungen `?` und `!` selbst keine Scheme-Ausdrücke sind und keine Werte zurückgeben, können in ihren Parameterlisten Scheme-Ausdrücke enthalten sein, die unifiziert werden müssen.

Da Scheme-Ausdrücke grundsätzlich (fast[1]) immer einen Wert zurückgeben (der auch unspezifiziert oder undefiniert sein kann, aber immer noch ein Wert ist) und alle Scheme-Werte mit der einzigen Ausnahme des speziellen Wertes `#f` (für „falsch“) bei Bedingungen als „wahr“ gelten, entfällt die Notwendigkeit für eine spezielle Auszeichnung von Guards.

Die Verwendung von *`values`* mit weniger oder mehr als einem Wert als Rückgabewert einer Transitionsanschrift ist unzulässig.

Damit werden *alle* Ausdrücke, die direkt an einer Transition (und nicht etwa an einer Kante oder Stelle) stehen zu Guards (bei anderen Formalismen für gefärbte Netze auch als „gate“ bezeichnet).

Guard

„**Guard**“ bezeichnet daher in einem Scheme-Referenznetz jeden Scheme-Ausdruck, der direkt an der Transition steht. (Auszeichnungen in geschweiften Klammern sind *keine* Scheme-Ausdrücke, also auch keine Guards.) Guards haben die Wirkung, dass die Transition nicht schalten kann, wenn der Ausdruck `#f` zurückliefert.

Wenn die Konjunktion aller Ausdrücke an der Transition also nach der Unifikation *nicht* den Wert `#f` hat kann die Transition schalten, wobei `action`-Anschriften zunächst immer einen unspezifizierten, von `#f` verschiedenen Wert zurück liefern und nicht alle Transitionsanschriften ausgewertet werden müssen, um diese Entscheidung zu treffen, da die Bindungssuche abgebrochen wird, so-

[1] Bei der Verwendung des Operators *`values`* können Ausdrücke genau genommen 0 oder mehr Werte zurückgeben. Nach R^{5}RS korrekte Scheme-Implementationen signalisieren bei `(if (`*`values`*`)#t #f)` einen Fehler. Derzeit (Juli 2007) gibt es noch einen Bug in SISC, durch den dieser Ausdruck aber `#t` liefert. Die Verwendung von *`values`* mit weniger oder mehr als einem Parameter an einer Transition ist ein Fehler, der aber wegen dieses Bugs nicht erkannt wird.

bald ein Guard `#f` liefert.

Neue Bezeichner an Transitionen

Als problematisch erweist sich die gelegentlich auftretende Notwendigkeit, neue Bezeichner in Beschriftungen einzuführen.

Bei einem frühen Entwurf des Verfahrens zur Bindungssuche war ich noch davon ausgegangen, dass die zur Verfügung stehenden Bibliotheken zur deklarativen Programmierung in Scheme dazu verwendet werden könnten, Variablen aus einer Multimenge von Werten (der Markierung einer Stelle) probeweise zu binden und dann anhand von Scheme-Ausdrücken (Guards) zu prüfen, ob die Bindung korrekt ist. Genau dieses Verfahren funktioniert in der Praxis auch, wenn die Bindung an der eingehenden Kante bereits vollständig ist. Ein Scheitern des Guards bringt in diesem Fall den Unifikationsalgorithmus von RENEW dazu, die bisherige Unifikation bis zur Bindung des Ausdrucks an der eingehenden Kante rückgängig zu machen, andernfalls bleibt die Bindung gültig und die Transition kann in diesem Modus schalten.

Wird ein neuer Bezeichner aber erst im Guard eingeführt (und nicht schon an der eingehenden Kante), kann der verwendete Unifikationsalgorithmus nicht allein anhand des Guard-Ausdrucks die korrekte Bindung des neuen Bezeichners erschließen.

Im Netz in Abbildung 7.1 auf der folgenden Seite wird an der eingehenden Kante der Transition mit der Beschriftung `(= xx x)` der Bezeichner `xx` an eine Marke gebunden (die nur den Wert `2` annehmen kann, aber das kann der Unifikationsalgorithmus nicht entscheiden). Man könnte meinen, der Ausdruck `(= xx x)` sorge nun für die Bindung des neuen Bezeichners `x` an den Wert des Bezeichners `xx`, aber das ist nicht der Fall. Aus Sicht der Unifikationsbibliothek[2] ist `=` eine Scheme-Variable ohne besondere Eigenschaften. Der Algorithmus könnte allenfalls noch in der Lage sein, mit dem Prädikat `procedure?` herauszufinden, dass es sich um eine

[2] Sowohl bei KANREN als auch SCHELOG (siehe Abschnitt 9.1 (S. 113)).

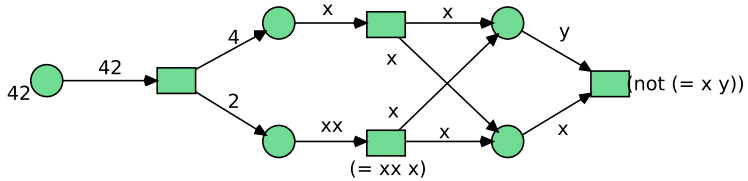

Abbildung 7.1: Das Netz „colored“. Falsche Übersetzung von Java nach Scheme

Prozedur handelt, aber wie sich diese verhält ließe sich nur herausfinden, indem alle möglichen Werte ausprobiert werden. Die Menge der möglichen Werte für `x` ist aber zum Zeitpunkt der Auswertung viel zu groß.

Nach dem R^{5}RS-Standard handelt es sich zwar bei `=` um ein Gleichheitsprädikat für Zahlen, aber das könnte durch eine einfache Deklaration im Deklarationsknoten – wie z. B. `(define (= x y)#f)` – zu einer ungültigen Annahme werden. Natürlich wird ein „anständiger Programmierer“ nicht ausgerechnet die Prozedur `=` in dieser Weise umdefinieren. Aber auch das ist wieder für keinen Algorithmus erkennbar. Die „Lösung“ Äquivalenzprädikate vor einer solchen Behandlung zu schützen, verbietet sich wiederum, da dadurch die Abstraktionsmöglichkeiten des Entwicklers eingeschränkt würden. (Beispielsweise wird `=` auch gerne als formaler Parameter in Prozeduren verwendet, die Gleichheitsprädikate als Argumente akzeptieren. Die Bindung von `=` an einer Kante könnte für einen ähnlichen Zweck sinnvoll sein.)

Scheme verfügt außerdem bereits im Standard über nicht weniger als sechs Prädikate die unterschiedliche Arten von Äquivalenz prüfen (die polymorphen `eq?`, `eqv?`, `equal?` (wobei `eq?` am stärksten und `equal?` am schwächsten diskriminiert), sowie die typspezifischen `=`, `string=?` und `string-ci=?`); einige `SRFI`-Dokumente und weitere Bibliotheken definieren zusätzliche Äquivalenzprädikate. Für jedes dieser Prädikate eine Sonderbehandlung zu implemen-

tieren wäre sowohl aufwändig als auch sehr unelegant.

Man könnte sich zwar mit dem Trick behelfen, dass man einen ungebundenen Bezeichner versuchsweise mit der anderen Seite des Ausdrucks unifiziert, das funktioniert aber nur unter sehr speziellen Randbedingungen mit einfachen Ausdrücken. Außerdem wäre dazu mindestens das Wissen erforderlich, dass es sich um ein Äquivalenzprädikat handeln soll.

Die einzig sinnvolle Lösung besteht also darin, einen speziellen Operator `is?` einzuführen, mit dem Gleichheitsspezifikationen in Scheme-Referenznetzen explizit gemacht werden können. Dieser Operator entspricht weitestgehend der Gleichheitsspezifikation in Java-Referenznetzen.

Da bestimmte Datentypen (z. B. Closures) in Scheme nicht immer sinnvoll miteinander verglichen, und damit auch nicht unifiziert werden können, wird neben dem allgemeinen Unifikationsoperator noch eine Art Definitionsoperator `bind` benötigt, mit dem solche Objekte an Bezeichner gebunden werden können. Der Effekt dieses Operators ist vergleichbar mit einer Zuweisung in einer `action`-Anschrift eines Java-Referenznetzes, mit einigen wichtigen Unterschieden.

Einführen von Bezeichnern mit `bind`

`syntax: (bind` *`variable expression`*`)`
$\Longrightarrow$ *`unspecified`*

Mit der Syntax `bind` kann der Bezeichner *`variable`* an den Wert des Ausdrucks *`expression`* gebunden werden. In *`expression`* dürfen weitere in der Umgebung ungebundene Bezeichner vorkommen, diese müssen dann aber anderweitig gebunden werden können (z. B. an eingehenden Kanten, oder weiteren `bind` oder `is?`-Ausdrücken).

Die Bindung erfolgt in der aktuellen sekundären Umgebung (so dass `bind` in einer Befehlssequenz verwendet werden kann) und auch so, dass die Bindung für alle späteren Auswertungen und Unifikationen des gleichen Schaltvorgangs zur Verfügung steht.

Der lexikalische Geltungsbereich wird also durch den `bind`-Operator durchbrochen, da eine (für den Schaltvorgang) globale Bindung erzeugt wird.

Implementiert wird `bind` durch ein Makro, welches zu einem *`define`*-Ausdruck expandiert wird. Dadurch wirkt `bind` im Gültigkeitsbereich des Ausdrucks selbst wie eine Deklaration mittels *`define`* und darf auch nur an Stellen vorkommen, an denen Definitionen legal sind, d. h. nur zu Anfang eines lexikalischen Gültigkeitsbereichs, wie etwa eines *`lambda`*- oder *`let`*-Ausdrucks.

Das folgende Beispiel bindet `y` an `10`, so dass `y` in anderen Beschriftungen der gleichen Transition verwendet werden kann und im folgenden Ausdruck `(> y x)` wird `y` dann verwendet (ohne diesen Ausdruck wäre der gesamte *`let`*-Ausdruck illegal):

```
(let ((x 6))
  (bind y (+ x 4)) ; y stays visible outside of the let-form
  (> y x))
```

Die Semantik des obigen Ausdrucks in der sekundären Umgebung selbst wäre am ehesten vergleichbar mit:

```
(define y)
(let ((x 6))
  (set! y (+ x 4))
  (> y x))
```

Dass `y` darüber hinaus noch in anderen Ausdrücken an der gleichen Transition sichtbar wird, ist konform mit der Festlegung, dass für die Bindungssuche an einer Transition nur *eine* sekundäre Umgebung verwendet wird. Der Mechanismus der Implementation im RENEW-Plugin beschränkt sich aber nicht auf die sekundäre Umgebung, sondern zusätzlich wird die Variable `y` im entsprechenden `VariableMapper`-Objekt für die RENEW-Unifikation sichtbar.

Im Unterschied zur Zuweisung innerhalb eines `action`-Statements handelt es sich um eine frühe Bindung, außerdem kann `bind` ausschließlich zum Erzeugen von Bindungen verwendet werden. (Eine Zuweisung innerhalb eines `action`-Statements eines Scheme-Referenznetzes wird dagegen zumeist *weitgehend* wirkungslos blei-

ben, es sei denn die Identitätsfunktion wurde als Übergangsfunktion gewählt.)

Wenn als erstes Argument kein Bezeichner angegeben wird, führt dies zu einem Syntaxfehler.

Eine zweimalige Zuweisung des gleichen Bezeichners mit `bind` an der gleichen Transition ist ein Fehler und führt während der Simulation zu einem Laufzeitfehler.

Der Versuch einen in der primären oder sekundären Umgebung bereits gebundenen Bezeichner mit `bind` zu binden, führt ebenfalls zu einem Laufzeitfehler.

Die Semantik des `bind`-Operators unterscheidet sich also signifikant sowohl von der der Operatoren *`define`* und *`set!`* als auch von denen jeglicher Äquivalenzprädikate, so dass davon abgesehen wurde, einen dieser Operatoren für Scheme-Referenznetze anders zu definieren oder etwa „*`define`*“ als Bestandteil des Operatornamens zu verwenden. Der Name `bind` wird weder vom Standard noch von einer nennenswert verbreiteten Bibliothek verwendet, so dass er nicht mit anderen Verwendungsweisen in Scheme assoziiert ist.

Explizite Unifikation mit `is?`

`syntax: (is?` *`left-expression right-expression`*`)`
$\Longrightarrow$ *`boolean`*

Der Operator `is?` fordert explizite Unifikation an. Beide Ausdrücke können beliebig komplex sein und ungebundene Variablen enthalten, die bei erfolgreicher Unifikation gebunden werden. Genau wie bei der Gleichheitsspezifikation = in Java-Referenznetzen, deren Entsprechung dieser Operator ist, ist keine Richtung der Auswertung vorgegeben.

Anders als `bind` liefert ein `is?`-Ausdruck einen Wert und kann an beliebiger Stelle verwendet werden.

Bei erfolgreicher Unifikation wird `#t` zurückgegeben, sonst `#f`. Ebenso wie bei `bind` bleiben die erzeugten Bindungen *außerhalb des statischen Gültigkeitsbereichs* während der gesamten weiteren Bindungssuche einer Transitionsinstanz gültig.

Auch kann `is?` innerhalb beliebig komplexer Ausdrücke verwendet werden (anders als die Gleichheitsspezifikation bei Java-Referenznetzen), z. B. ist folgender Ausdruck erlaubt:

```
(and (is? (cons col 'left) sock1)
     (is? sock2 (cons col 'right)))
```

(z. Z. aber noch sehr ineffizient).

Die Unifikation wird (allgemein in allen Anschriften) aber nur als gültig betrachtet, wenn keine anonymen Variablen mehr im gefundenen Unifikator vorhanden sind. Der obenstehende Ausdruck würde, z. B. bei Verwendung des Logiksystems KANREN ([KANREN, 2006]) den Unifikator

```
((sock1.0 (_.0 . left))
 (sock2.0 (_.0 . right))
 (col.0 _.0))
```

liefern, sofern nicht entweder `col` oder sowohl `sock1` als auch `sock2` außerhalb des Ausdrucks gebunden werden (was natürlich in anderen Ausdrücken oder eingehenden Kanten erfolgen kann).
Dieser Unifikator muss vom Algorithmus zur Bindungssuche als *Endergebnis* der Unifikation abgelehnt werden, da die anonyme Logikvariable `_.0` keinen konkreten Wert angibt (sie entspricht den Objekten der Klasse `Unknown` im Unifikationsalgorithmus von RENEW). Als Zwischenergebnis ist er natürlich akzeptabel, sofern ein Verfahren implementiert wird, mit dem dieses Zwischenergebnis (z. B. durch Unifikation von `_.0` mit einem Wert) konkretisiert werden kann.

Die Art der Ausdrücke auf beiden Seiten ist nur durch die Möglichkeiten der verwendeten Logikbibliothek begrenzt.

Mit Einschränkungen ist es also auch möglich, dass beide Seiten Unbekannte enthalten, beispielsweise bindet

```
(is? (cons x 'b) (cons 'a y))
```

die Variable `x` an den Wert `a` und die Variable `y` an den Wert `b`.

Bereits gebundene Bezeichner können mehrfach in `is?`-Ausdrücken verwendet werden, bei einer Inkonsistenz wird die Unifikation fehlschlagen und der `is?`-Ausdruck liefert `#f` zurück.

Auf den ersten Blick scheint

```
(not (is? x 5))
```

ein interessanter Fall zu sein, der aber faktisch nicht problematisch ist. Der Bezeichner `x` (sofern ungebunden) wird an den Wert `5` gebunden. Sofern es sich um einen Guard-Ausdruck handelt, kann anschließend die Transition *nicht* schalten.

Bei einem Ausdruck an einer Kante würde `#f` aus einer Stelle entnommen oder in die Ausgangsstelle gelegt und `x` wäre für die gleiche Bindungssuche erfolgreich gebunden, was eine etwas umständliche Ausdrucksweise für eine andere Kantenbeschriftung wäre. Falls `x` in einem negierten `is?`-Ausdruck *nicht* gebunden werden könnte, würde die gesamte Unifikation fehlschlagen und die Transition könnte ebenfalls (mangels Bindungen für ungebundene Variablen) nicht schalten. Ob die ungebundenen Variablen im `is?`-Ausdruck irgendwo anders an der selben Transition verwendet werden, ist dabei irrelevant.

Das Netz `not-is` in Abbildung 7.2 auf der folgenden Seite illustriert die verschiedenen Möglichkeiten. Rechts daneben sieht man das Endergebnis eines Simulationslaufs. Im obersten Teilnetz ist die Transition in der Anfangsmarkierung nicht aktiviert, weil der Guard-Ausdruck `#f` liefert, und ebenso bei den unteren beiden, weil die Unifikation keine Bindungen geliefert hat.

Auf Grund dieser etwas verwirrenden Semantik sollten `is?`-Ausdrücke besser nicht negiert verwendet werden.

Interessant könnte die Verwendung von `is?` in Verbindung mit komplexen Logikausdrücken an Guards sein, leider ist bei dem derzeitigen Stand der Implementation der `is?`-Operator sehr ineffizient, weswegen von solchen Konstrukten momentan noch abzuraten ist.

Der Name des Operators wurde in Anlehnung an das Unifikationsprädikat `%is` des Scheme-Logiksystems SCHELOG gewählt. Von den alternativen Notationen == (KANREN) bzw. `%==` (Identitätsrelation in SCHELOG) wurde wegen der Verwechslung mit Äquivalenzprädikaten wie Javas == und Schemes = sowie der Gleichheits-

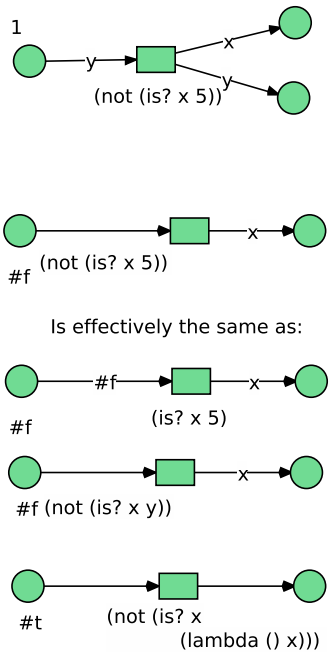

Abbildung 7.2: Negiertes `is?`

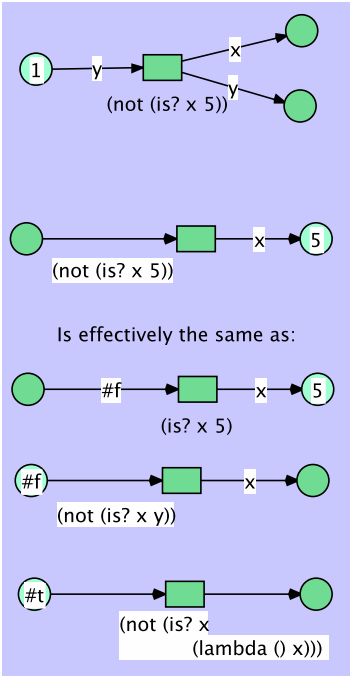

Abbildung 7.3: Ergebnis

spezifikation = der Java-Referenznetze abgesehen. Eine mögliche Benennung wäre noch `unify` gewesen, aber die verwendete Unifikationsprozedur in KANREN heißt bereits `unify`, so dass hier konkret eine Kollision vorkommen kann, wenn KANREN-Operatoren in Scheme-Referenznetzen verwendet werden sollen. Auch bestehen Kollisionsmöglichkeiten mit den bereits erwähnten Operatoren von KANREN oder SCHELOG.

Die Ähnlichkeit zur numerischen Bindung `is` in Prolog wird aufgrund der anderen, schwereren Verwechslungsmöglichkeiten in Kauf genommen.

Das Fragezeichen wurde wegen des booleschen Rückgabewertes hinzugefügt (die entsprechenden Relationen der Logiksysteme liefern dagegen Ziele („Goals“) zurück). Dass die Bezeichner für Scheme-Prädikate die entweder `#t` oder `#f` als Rückgabewert liefern, auf '`?`' enden, ist eine weit verbreitete Konvention.

7.5.2 Manuelle Transitionen

```
special: {manual}
```

Transitionen mit der Auszeichnung `{manual}` können nur dann schalten, wenn sie vom Benutzer angeklickt werden.

Beim Übersetzen und Simulieren des Netzes muss darauf geachtet werden, dass Transitionen mit dem Ausdruck `{manual}` genau so behandelt werden wie Transitionen in Java-Referenznetzen mit dem Schlüsselwort `manual`.

Vergleiche Kapitel 5 (S. 55).

7.5.3 Netzexemplare

Netzexemplare werden mit dem Operator `new` erzeugt.

```
procedure:  (new net-type)
                ⟹ net-instance
```

Die Prozedur `new` erzeugt eine Netzinstanz (ein in ein Scheme-Objekt gekapseltes Java-Objekt vom Typ `NetInstance`) nach dem Netzmuster des Namens `net-type` (ein Scheme-Symbol).

Da `new` eine Prozedur bezeichnet, kann `net-type` durch einen beliebigen Ausdruck spezifiziert werden, es ist also möglich, er-

zeugte Netzexemplare z. B. durch Marken zu parametrisieren. Synchrone Kanäle können aber absichtlich nicht in dieser Weise parametrisiert werden, da am Namen des Kanals die Lokalität der Transition direkt ablesbar sein soll.

Der Aufruf von `new` kann an jeder beliebigen Stelle erfolgen, also auch in der initialen Markierung einer Stelle. Dabei ist Vorsicht geboten, denn wenn ein Netzexemplar *des gleichen Typs* wie das aktuelle Netz erzeugt werden soll, führt dies in einer Stelle zwingend zu einer Endlosschleife, die zur Erschöpfung des Heaps (und damit in RENEW zu einem `OutOfMemoryError`) führt. Auch kann die Verwendung von `new` an eine Kante ebenfalls zu einer solchen Endlosschleife führen, sofern der Ausdruck bereits beim Erzeugen des Netzes ausgewertet wird.

Da Netzmuster Namen haben können, die Großbuchstaben und Leerzeichen enthalten können, welche in Scheme-Bezeichnern nach [R^{5}RS, 1998] ungültig sind, muss hier unter Umständen die spezielle Reader-Syntax verwendet werden, die in SISC auch für andere Java-Namen verwendet werden kann, indem Symbole mit Großbuchstaben, Leer- und Sonderzeichen in senkrechte Striche eingeschlossen werden, wie z. B. `'|NetzMuster|`.

Es gibt, im Unterschied zu Java-Referenznetzen, für die Bindung einer Variablen an ein Netzexemplar keine spezielle Syntax. Wie jedes andere Scheme-Objekt ist ein Netzexemplar zunächst namenlos und wird z. B. durch einen `bind`-Ausdruck an der Transition gebunden.

Außerdem wird auch keine spezielle Syntax für den Aufruf eines synchronen Kanals bei der Erzeugung des Netzexemplars angeboten. Nach der Erzeugung des Netzexemplars kann es durch Unifikation an einen Bezeichner gebunden werden und jeder Kanal kann an einer Transition mit dem Operator `!` aufgerufen werden. Es gibt keinen speziellen Kanal `new`, ein solcher kann aber explizit definiert und aufgerufen werden.

Das Verhalten der Java-Beschriftung `net:`*`new`*` NetName()`, die ein Netzexemplar des Musters `NetName` an einer Transition erzeugt, an die Variable `net` bindet und den Kanal *`new`* ohne Parameter auf-

ruft, lässt sich in einem Scheme-Referenznetz wie folgt nachbilden:

```
(bind net (new '|NetName|))
{! new net}
```

Hier kann aber statt `new` auch ein anderer, möglichst treffender, Name für den Kanal verwendet werden.

In dieser Notation könnten prinzipiell auch außerhalb des RENEW-Simulators Netzexemplare in Scheme verwendet werden, so wie dies mit Java-Referenznetzen möglich ist. Die konkrete Umsetzung soll aber nicht Thema dieser Arbeit sein.

Die Verwendung von Netzexemplaren als Scheme-Objekte ist in SISC durch die enge Java-Integration möglich, die die Verwendung von Java-Objekten in Scheme-Code erlaubt. Wie bereits in Kapitel 5 (S. 55) erwähnt, werden keine Scheme-Operatoren zur Arbeit mit synchronen Kanälen bereitgestellt, es wäre aber durchaus möglich, mit Hilfe des Java-Interfaces von SISC solche zu implementieren. Durch deren „versteckte“ Verwendung in benutzerdefinierten Operatoren würde dann die in Abschnitt 1.3.4 (S. 13) aufgestellte Forderung verletzt, dass die Lokalität einer Transition direkt an der Beschriftung erkennbar sein soll. Verhindern ließe sich dies durch einen am Ende von Kapitel 6 angedeuteten speziellen Formalismus, in dem die Bindungen des SISC-Java-Interfaces in der primären Umgebung fehlen.

Der folgende Unterabschnitt beschreibt die zum vorgestellten Lokalitätsbegriff konforme Notation für synchrone Kanäle.

7.5.4 Synchrone Kanäle

Downlinks werden mit `!` aufgerufen, Uplinks mit `?` erzeugt.

`special: {! link-name net parameter ...}`

`special: {? link-name parameter ...}`

`procedure: (this)` $\Longrightarrow$ *`current-net-instance`*

Der Operator `?` stellt an der betreffenden Transition den Uplink mit Namen *`link-name`* bereit und der Operator `!` ruft den

Link *link-name* im Netzexemplar *net* auf (beim Scheme-Plugin für RENEW ein in Scheme gekapseltes Java-Objekt vom Typ `NetInstance`). Bei der Unifikation wird dafür gesorgt, dass die beim Uplink und beim Downlink angegebenen Parameter miteinander unifiziert werden, siehe Kapitel 9 (S. 113).

Der Parameter *link-name* *muss* ein *literales* Symbol sein (ohne `quote` oder `'`, ein Ausdruck kann *nicht* verwendet werden), damit der Name des Kanals eindeutig feststeht. Andernfalls wäre die Lokalität der Transition u. U. variabel und nicht mehr an der Beschriftung ablesbar. Außerdem ist es bei RENEW nur beim Übersetzen des Netzes möglich den Kanalnamen festzulegen.

Wie alle anderen speziellen Auszeichnungen haben `?` und `!` keinen Rückgabewert und werden nur nach den in Kapitel 5 (S. 55) angegebenen Regeln wirksam.

Die Verwendung von synchronen Kanälen außerhalb von Transitionsanschriften ist nicht möglich. Die geschweiften Klammern in SISC sind eigentlich legale Bestandteile von Bezeichnern, weswegen ihre Verwendung in den meisten Fällen keinen Syntaxfehler erzeugen wird. Beispielsweise ist `{! ch (this)a b}` an einer Kante eine vierfache Kantenbeschriftung aus dem aktuellen Netzexemplar `(this)` und den (möglicherweise ungebundenen) Bezeichnern `{!` (eine offene geschweifte Klammer gefolgt von einem Ausrufezeichen, kein Tippfehler sondern ein legaler Bezeichner in SISC), `ch`, `a` und `b}` (sic!). Es erscheint daher sinnvoll für die Verwendung in Scheme-Referenznetzen die erlaubten Bezeichner so weit einzuschränken, dass die in R^{5}RS reservierten Zeichen `{}` nicht mehr als Namensbestandteile verwendet werden können.

Wie bei `bind` und `is?` erfolgt die Bindung für die gesamte weitere Bindungssuche an der Transition.

Damit ein Netzexemplar auch einen Kanal zu sich selbst aufrufen kann, kann an Stelle des Parameters *net* bei dem Downlink-Operator `!` auch die Prozedur `this`[3] verwendet werden, die das aktuelle Netzexemplar selbst zurück gibt.

[3] Dank an Michael Duvigneau für diese Anregung.

Der Aufruf `(this)` kann, anders als `?` und `!`, nicht nur an Transitionen sondern überall erfolgen, z. B. auch in der initalen Markierung einer Stelle. Es wird lediglich eine Referenz auf das aktuelle Netzexemplar angelegt, was nicht zu einer Endlosschleife führen sollte. Das Netzexemplar wird damit zu einer zirkulären Datenstruktur, was für naiv implementierten Programmcode, der damit arbeiten soll, zum Problem werden kann.

Wie bei Netzexemplaren in Java-Referenzetzen kann es in einem Netz mehrere mögliche Uplinks geben (aber nur einen pro Transition), von denen per Zufall einer ausgewählt wird.

Die Operatoren `!` und `?`, werden von der Scheme-Bibliothek TERMITE[4] verwendet, wo sie ähnlich wie bei CSP[5] für das Absenden (`!`) oder das Empfangen (`?`) einer Nachricht stehen. Da es bei synchronen Kanälen im Gegensatz dazu keine festgelegte Richtung der Informationsübermittlung gibt, besteht hier eine gewisse Verwechslungsgefahr.

Diese Gefahr ist aber gering, da zum Einen die Operatoren bei CSP in Infix-Notation verwendet werden, zum Anderen dürften Scheme-Referenznetze kaum im Zusammenspiel mit TERMITE oder ähnlichen Synchronisationsbibliotheken verwendet werden, da ihre synchronen Kanäle bereits diverse Möglichkeiten zur Modellierung verteilter Systeme bieten.

Außerdem wird durch `!` der aktive, die Kommunikation initiierende Part gekennzeichnet, da für den Aufruf eines Downlinks ein Netzexemplar als Marke gebunden sein muss, während `?` den passiven Kommunikationspartner auszeichnet, wie es auch in den Bezeichnungen „Downlink“ und „Uplink“ mitschwingt. Außerdem bietet die Notation mit `!` bzw. `?` den Vorteil, dass der Name des Kanals optisch hervorgehoben wird, aber die Zeichen trotz ihrer Kürze jedem geübten Leser von in lateinischer Schrift geschriebenen Sprachen deutlich ins Auge fallen. Sind Uplink- und Downlink-Beschriftungen untereinander angeordnet, dann stehen die Kanal-

[4] Erhältlich unter `http://toute.ca/`

[5] [Hoare, 1978]

namen direkt untereinander, was bei einer Einbettung in Scheme-Programmtexte übersichtlicher sein dürfte.

Zusätzlich könnten zwei Operatoren `downlink` und `uplink` als Synonyme hinzutreten, aber diese Namen sind an sich genau so erklärungsbedürftig wie die gewählten Operatorenbezeichnungen und es scheint darum unökonomisch und unästhetisch, zwei optionale Namen für die gleiche Operation einzuführen.

Die Notation für synchrone Kanäle ist damit ähnlich knapp wie bei Java-Referenznetzen, was bei der grafischen Erstellung von Scheme-Referenznetzen einen weiteren Vorteil bedeutet.

7.5.5 Nicht-Unifizierbare Transitionsanschriften

Actions

Ausdrücke an Transitionen, die innerhalb des Körpers eines `action`-Formulars stehen (**Actions**), werden verspätet ausgewertet. Darum kann man auch von späten Transitionsanschriften sprechen.

```
syntax:  (action expression1 ...)
            ⟹ unspecified
```

Außerhalb einer Transition oder Kante führt die Verwendung des `action`-Operators zu einem Laufzeitfehler, was in RENEW dadurch begründet ist, dass kein `CalculationChecker`-Objekt zur Verfügung steht, bei dem die Auswertung angemeldet werden könnte. Es ist unklar, warum eine Aktion in einem Deklarationsknoten oder einer Stelle verwendet werden sollte. Darum erscheint diese Einschränkung auch für eine Einbettung in Scheme sinnvoll.

Die Auswertung des Körpers geschieht konzeptionell wie folgt:

1. Der gesamte Körper wird bereits vor der Auswertung (während der Unifikation) mittels `delay` und *`begin`* in ein „Versprechen“ (Promise) verpackt (siehe [R^{5}RS, 1998]). Aus diesem Grund muss `action` zwingend als Makro implementiert werden.

2. Die Aktion wird an eine der Transitionsinstanz zugeordnete Liste (nicht notwendigerweise ein Scheme-Datenobjekt) angehängt.

3. Nach erfolgter Unifikation wird beim Schalten der Transition diese Liste abgearbeitet und jedes einzelne Element wird mittels `force` in der aktuellen sekundären Umgebung der Transition evaluiert.

(Praktisch erfolgt die Auswertung von `action` Anschriften in Java-Referenznetzen mit Hilfe der `CalculationChecker`-Objekte. Es liegt daher nahe, diese auch zum Registrieren der geplanten Aktionen in Scheme-Referenznetzen zu verwenden.)

Im Unterschied zum `action`-Schlüsselwort in Java-Referenznetzen findet hier keine Analyse von Abhängigkeiten statt. Sofern eine Auswertungsreihenfolge von Bedeutung ist, sollten die entsprechenden Ausdrücke zusammen in einen einzigen `action`-Aufruf verpackt werden.

Es können auch keine Bindungen in Actions eingeführt werden, diese Möglichkeit besteht allein mit den speziellen Bindungsoperatoren.

Streng genommen sind die Action-Beschriftungen in Scheme-Referenznetzen auch unifizierbar; der Wert des Ausdrucks selbst ist immer unspezifiziert aber verschieden von `#f`, wird also als erfolgreicher Guard-Ausdruck gewertet. Die Liste der Aktionen selbst ist in der sekundären Umgebung nicht zugänglich, damit nicht schon während der Unifikation mit `force` die Auswertung einer Aktion erzwungen werden kann.

Damit kann die Aktion faktisch keinen Einfluss auf die Unifikation ausüben.

Da Actions lediglich ein Mechanismus sind, um den Zeitpunkt der Wirksamkeit eines Nebeneffektes zu kontrollieren, besteht kein Grund warum diese als spezielle Auszeichnungen implementiert werden sollten. Das Schlüsselwort `action` bietet auch in Java-Referenznetzen keinerlei Hinweis darauf, ob ein Nebeneffekt stattfindet oder nicht, wie an den Beispielen aus Abschnitt 1.3 (S. 7) zu sehen ist. Damit steht der Verwendung von Actions in Makros oder Prozeduren nichts im Wege. Es ist aber empfehlenswert in solchen Bindungen „`action`“ als Bestandteil des Namens zu verwenden,

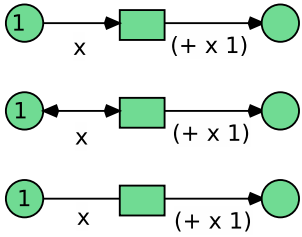

Abbildung 7.4: Von oben nach unten: Normale (Fluss-) Kante, Reserve- und Test-Kante

damit erkennbar bleibt, dass durch ihre Auswertung Nebeneffekte registriert werden.

7.6 Kantentypen

In diesem Abschnitt werden noch die Besonderheiten einiger Kantentypen erklärt.

7.6.1 Gewöhnliche Kanten, Testkanten und Reservekanten

Diese Kantentypen zeigen kein spezielles Verhalten bei der Unifikation. Reservekanten sind lediglich Abkürzungen für doppelte Kanten mit gleicher Beschriftung. Testkanten binden, ohne eine Marke zu entfernen. Semantisch ergibt sich keine Differenz zu Java-Referenznetzen.

7.6.2 Inhibitorkanten

Wie bei Java-Referenznetzen wird hier bei der Unifikation die Bedeutung der Beschriftung (sofern vorhanden) umgedreht. Ist keine Beschriftung vorhanden, kann die Transition nur schalten, wenn die

Stelle leer ist. Wenn eine Beschriftung vorhanden ist, kann nur geschaltet werden, wenn die Markierung *nicht* mit der Beschriftung unifizierbar ist. Die Beschriftung darf keine ungebundenen Variablen enthalten, da aus der Markierung der Stelle keine Information über die Variablenbindung generiert werden kann.

Diese Kantenart steht wegen ihrer unklaren Nebenläufigkeitssemantik wie bei Java-Referenznetzen nur im sequentiellen Simulationsmodus zur Verfügung.

Unter Umständen könnten spätere Forschungsergebnisse eine Nebenläufigkeitssemantik für Inhibitorkanten in Referenznetzen erbringen, so dass diese auch im nebenläufigen Simulationsmodus eingesetzt werden könnten. Möglicherweise lassen sich dafür Ansätze aus [Baldan u. a., 2000] verwenden.

7.6.3 Flush-Kanten

Diese Kantenart (auch „Clear-Arc“) entfernt alle Marken aus einer Stelle. Findet sich eine Beschriftung an der Kante, so wird die Multimenge der Marken an eine Scheme-Liste gebunden und mit der Beschriftung unifiziert. Wie bei Java-Referenznetzen wird auch mit einer leeren Markierung unifiziert und in diesem Falle die leere Liste `()` gebunden.

Diese Kantenart steht wegen ihrer unklaren Nebenläufigkeitssemantik wie bei Java-Referenznetzen nur im sequentiellen Simulationsmodus zur Verfügung.

7.6.4 Flexible Kanten

Ähnlich wie bei den Flush-Kanten wird die Beschriftung als Scheme-Liste aufgefasst. Eingehende flexible Kanten liefern aber wie Inhibitor-Kanten keine Information über ihre Bindungen, diese kann aber z. B. mit einem `bind`-Ausdruck an der Transition oder an einer Test-Kante geliefert werden. Eingehende flexible Kanten unifizieren jedes Element der an ihre Anschrift gebundenen Liste mit

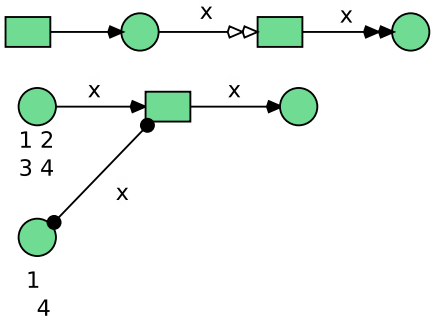

Abbildung 7.5: Oben: Flush-Kante gefolgt von flexibler Kante, unten: Inhibitorkante

einer Marke der Stelle, ausgehende lösen die an ihre Beschriftung gebundene Liste in eine Multimenge von Marken auf.

Im Beispiel in Abbildung 7.6 rechts gegenüberliegend (übertragen aus [Kindler und Völzer, 1997, S. 347]) wird in einem Netzwerk von Agenten für jeden Agenten festgestellt, was sein kürzester Abstand zu einem Wurzelagenten (der Quelle des Netzwerks) ist. Das Endergebnis ist die Multimenge aus Paaren von Agent und Distanz in der Stelle „distance“.

Das Beispiel in Abbildung 7.7 auf der folgenden Seite ist aus [Kummer u. a., 2006b] entlehnt und zeigt beide Arten von flexiblen Kanten.

Sowohl bei Flush- als auch bei flexiblen Kanten könnte man alternativ über eine Unifikation mit Vektoren statt Listen nachdenken. Dabei unterstützt aber nur SCHELOG als eines der beiden in Frage kommenden Logiksysteme den Scheme-Datentyp Vektor, vermittelt durch die Funktionen `vector->list` und `list->vector`. Dies ließe sich in der Art zwar auch für KANREN leicht nachrüsten, damit wären dann aber auch Überlegungen über etwaige Performanzgewinne durch den Gebrauch von Vektoren statt Listen hinfällig.

Der Umstand, dass flexible Eingangskanten keine Information

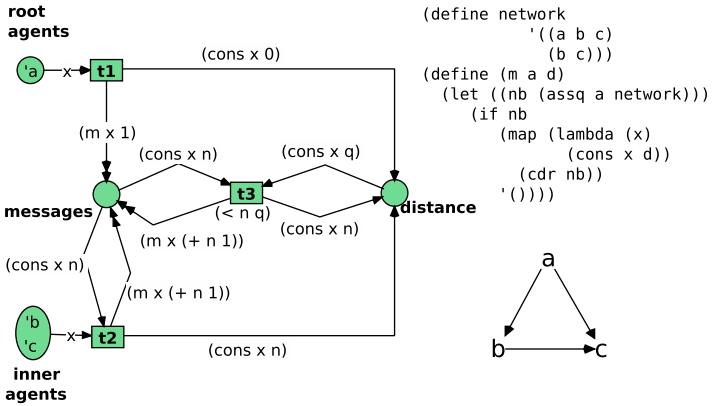

Abbildung 7.6: Geringster Abstand zu einer Quelle eines Graphen

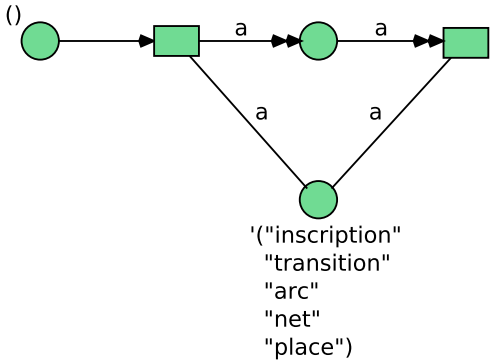

Abbildung 7.7: Flexible Eingangs- und Ausgangskanten

über mögliche Bindungen generieren können, wird in [Kummer, 2002] teilweise mit Beschränkungen der Programmiersprache Java begründet, die seit Java Version 5 nicht mehr zutreffen, wie z. B. fehlendem parametrischen Polymorphismus.

Für eine Anwendung in Scheme-Referenznetzen wäre zu überlegen, ob sich in flexiblen Eingangskanten z. B. List-Comprehensions (nach [SRFI-42, 2003]) oder andere Mechanismen einsetzen lassen, mit denen eine Untermenge der Markierung ausgefiltert werden kann.

Eine präzise Spezifikation eines solchen Mechanismus wird aber dadurch erschwert, dass normalerweise eine Marke oder im Fall von flexiblen Kanten und Flush-Kanten eine Multimenge von Marken in einem einzigen Schritt mit der *gesamten* Beschriftung der Kante unifiziert wird. Eine Filterung von Marken müsste aber in zwei Schritten erfolgen: dem Ausdruck müssten zunächst alle Marken in der Stelle zur Verfügung stehen aus denen im zweiten Schritt die gewünschten ausgefiltert werden könnten.

Für den Fall, dass die Beschriftung einer eingehenden flexiblen Kante in der ersten Phase zu einer Prozedur mit einem Parameter ausgewertet wird, könnte diese Prozedur dann in der zweiten Phase auf die Liste aller Marken in der Stelle angewendet werden, um letztendlich die gewünschte Teilmarkierung zu erhalten. Es ist fraglich, ob so eine Auswertung in zwei Phasen noch der Spezifikation der flexiblen Kanten entspricht oder ob hier schon eine ganz neue Kantenart beschrieben wird.

In der Praxis reicht eine Kombination aus Flush-Kanten und flexiblen Kanten aber aus, um einen hinreichend ähnlichen Effekt zu erreichen: Eine Flush-Kante entfernt alle Marken aus der Stelle und bindet sie an eine Liste, die dann in den Ausdrücken von flexiblen Ausgangskanten gefiltert oder anderweitig umgerechnet werden kann.

8 Beispiele für Scheme-Referenznetze: Konkretisierung des Entwurfs

In diesem Kapitel werden die im vorigen Kapitel grob umrissenen konzeptionellen Stufen und die dort vorgestellten Netzanschriften aufgegriffen, um den Entwurf mit Hilfe von Beispielen zu konkretisieren. Abschließend sollen weitere Beispiele die Möglichkeiten der Scheme-Referenznetze illustrieren.

8.1 Konzeptionelle Stufen

In Abschnitt 7.1 (S. 65) wurde angedeutet, dass sich „Scheme-Coloured-Nets“ als eine konzeptionelle Vorstufe zu Scheme-Referenznetzen auffassen lassen. Anhand der vorgestellten Anschriftstypen lässt sich diese konzeptionelle Unterteilung konkretisieren und verfeinern.

Diese Stufen dienten bei der Implementation als Richtschnur. Da die meisten Stufen auf den vorhergehenden aufbauen, es aber keine zyklischen Abhängigkeiten gibt und alle positiven Eigenschaften der vorigen Stufe in der nächsten beibehalten werden, ergab sich aus ihnen eine klare Abfolge von Meilensteinen, die nacheinander realisiert und getestet werden konnten, bevor zur nächsten Stufe übergegangen wurde.

Alle Stufen erlauben die Verwendung beliebiger Scheme-Ausdrücke als Beschriftungen, dabei sind zunächst gar keine Ausdrücke an Transitionen erlaubt.

8.1.1 Netze mit Scheme-Objekten in Stellen und an Kanten

Dies ist die einfachste Vorstufe, bei der die Marken aus beliebigen Scheme-Objekten bestehen können und lediglich Scheme-Ausdrücke als Kantenbeschriftungen erlaubt sind, die keinerlei ungebundene Bezeichner haben. Die Ausdrücke an den Kanten müssen bei Auswertung Objekte liefern. Wenn eine Marke in einer Stelle nach dem Scheme-Prädikat `equal?` mit dem Ergebnis eines Kantenausdrucks übereinstimmt, kann die Transition schalten.

Diese Stufe ist vergleichbar mit den „Arc-Constant (Coloured) Nets“ aus [Girault und Valk, 2003], mit dem wichtigen Unterschied, dass die Ausdrücke an den Kanten nicht zwingend konstante Werte annehmen müssen, da hier bereits Nebeneffekte oder nicht-definite Prozeduren verwendet werden können, die dazu führen dass ein Kantenausdruck bei jeder Auswertung einen unterschiedlichen Wert liefert.

Als „Black-Token“ wird die leere Liste verwendet. Bei unbeschrifteten Kanten wird davon ausgegangen, dass sie mit der leeren Liste `()` beschriftet sind.

Der Deklarationsknoten kann bereits verwendet werden.

Das Netz in Abbildung 8.1 auf der folgenden Seite ist paradigmatisch für diesen einfachen Formalismus. Im Deklarationsknoten unten rechts wird der Bezeichner `x` mit dem Wert `5` deklariert, so dass die Stelle links oben in ihrer Anfangsmarkierung *zweimal* eine Marke mit dem Wert `5` erhält.

Die Kanten mit der Beschriftung `x` entfernen dann auch eins dieser Objekte aus der Stelle bzw. legen ein neues Objekt mit dem gleichen Wert als Marke in die Ausgangsstelle.

Die Beschriftung `'(a b c)` erzeugt eine Liste `(a b c)`. Alle mit diesem Ausdruck erzeugten Listen im Netz sind zueinander `equal?`, aber *nicht* `eq?` oder `eqv?`.

Eine wichtige Entwurfsentscheidung auf dieser Stufe ist also, das Prädikat `equal?` zu verwenden, um Marken bzw. *Werte* an Kanten miteinander zu vergleichen.

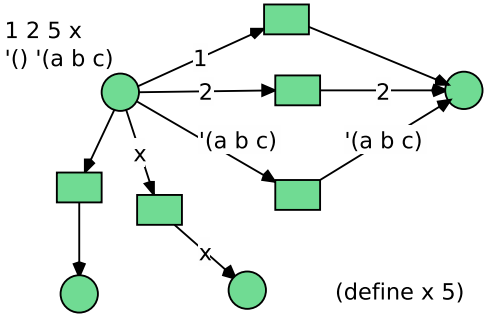

Abbildung 8.1: Scheme-Objekte an Kanten und Transitionen

Die Verwendung von `eqv?` oder gar `eq?` würde dazu führen, dass Listen nicht sinnvoll als Marken verwendet werden könnten, da diese Prädikate dafür zu stark diskriminieren:

```
(eqv? '(a b c) '(a b c))
⟹ #f
```

8.1.2 Scheme-Objekte mit Pattern-Matching an Kanten

In der nächsten Stufe werden ungebundene Bezeichner an Kantenbeschriftungen erlaubt. Transitionsbeschriftungen sind weiterhin nicht erlaubt (bzw. können ignoriert werden).

Die Ausdrücke an den eingehenden Kanten werden mittels Pattern-Matching gebunden, wobei die ungebundenen Bezeichner geeignete Werte erhalten. Auf dieser Stufe muss bereits eine geeignete Logikbibliothek eingebunden werden, die später auch zur Unifikation verwendet werden kann.

Das Netz `rota-simple` in Abbildung 8.2 auf der nächsten Seite erhält in der einzigen Stelle die Liste `(r o t a)` als Anfangsmarkierung. Das Pattern `(cons a e)` an der Eingangskante der Transition kann erfüllt werden, indem `a` an das erste Element der Liste und `e` an den Rest der Liste gebunden werden. An der Ausgangskante der Transition wird dann `(a)` an die Liste `e` angehängt, so dass

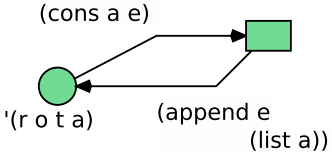

Abbildung 8.2: Pattern-Matching an Kanten

die neue Marke die Rotation der alten Marke wird. Aus `(r o t a)` wird nach dem ersten Schaltvorgang `(o t a r)`.

Voraussetzung für den Erfolg des Pattern-Matching ist, dass der Unifikationsalgorithmus mit den Ausdrücken und Werten umgehen kann. Sowohl mit SCHELOG als auch mit KANREN kann beispielsweise `5` mit `x` in Übereinstimmung gebracht werden, aber nicht mit `(+ x 1)`. Operatoren, die eine dynamische Typprüfung vornehmen (wie der hier verwendete Operator +, aber auch =, *append*, u. v. m.), können *generell nicht* in Patterns verwendet werden.

Wenn man imperative oder auch funktionale Programmierung gewöhnt ist, lesen sich die Ausdrücke zunächst ungewohnt. Normalerweise würde mit dem Ausdruck `(cons a e)` ein Paar aus den Werten der Variablen `a` und `e` gebildet werden, als Pattern verstanden scheint der Ausdruck quasi *rückwärts* ausgewertet zu werden: aus einer konkreten Liste werden Kopf und Rest erhalten. Es zeigt sich in Abschnitt 9.3 (S. 116), dass in einem ähnlichen Fall tatsächlich *vor* der Unifikation zunächst ein Paar aus den Unbekannten gebildet wird.

Ab dieser Stufe können auch einige einfache Beispiele für Java-Referenznetze in Scheme-Petrinetze übertragen werden:

In den beiden Beispielen in Abbildung 8.3 rechts gegenüberliegend und Abbildung 8.4 auf der folgenden Seite werden zwei unterschiedliche Möglichkeiten verdeutlicht, Beispiele zur Unifikation von Tupeln bzw. Listen auf dieser Stufe streng nach Scheme zu übersetzen. Beide Beispiele verwenden Paare (Listen sind in Scheme aus Paaren, so genannten Cons-Zellen, zusammengesetzt),

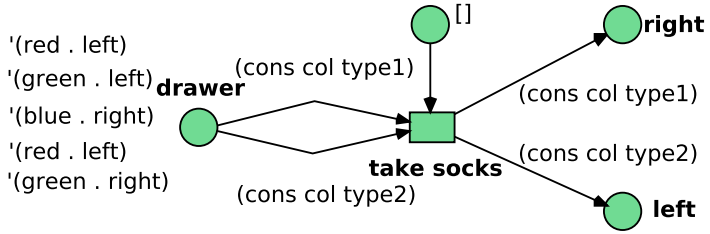

Abbildung 8.3: Das Netz „socks“

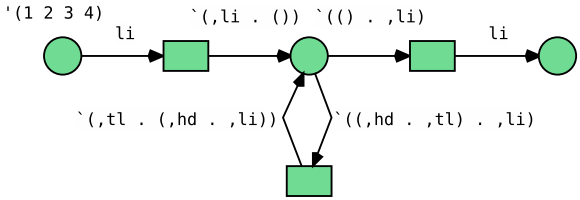

Abbildung 8.4: Das Netz „reverse-qq“

wobei im ersten Beispiel das Paar mit den Operator `cons` erzeugt wird, und im zweiten die Operatoren `quasiquote` (mit dem Reader-Makro `` ` ``)[1] und `unquote` (Reader-Makro: `,`) verwendet werden, um eine kürzere (nicht unbedingt besser lesbare) Darstellung der zu unifizierenden Listen zu erreichen.

8.1.3 Scheme-Coloured-Nets

Die erste Stufe von Petri-Netzen mit Scheme-Beschriftungen, die als gefärbte Netze angesehen können, fügen zur vorhergehenden Transitionsbeschriftungen hinzu, die als Guards (oder auch Gates) betrachtet werden.

Für die erfolgreiche Auswertung eines Guards müssen in dieser

[1] nicht zu verwechseln mit dem Operator `quote` mit dem Reader-Makro `'`

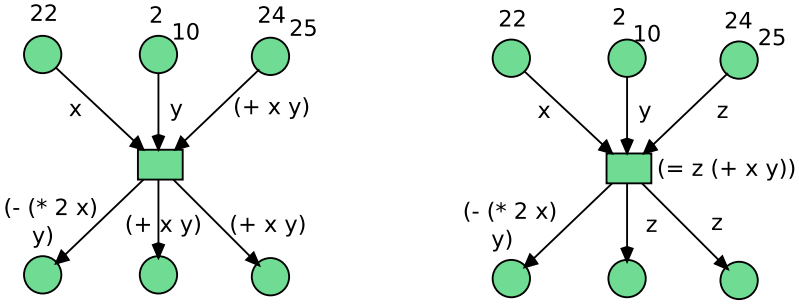

Abbildung 8.5: Das Netz „equality“

Stufe alle Bezeichner im Guard-Ausdruck gebunden sein.

Das Netz in Abbildung 8.5 zeigt zwei unterschiedliche Ansätze zur Unifikation, einmal über Ausdrücke mit den Bezeichnern der eingehenden Kanten (links) und andererseits durch einen booleschen Ausdruck an der Transition.

Das linke Teilnetz gehört dabei noch zur vorhergehenden Stufe, da kein Guard verwendet wird; im rechten wird dagegen ein Guard-Ausdruck verwendet, aber alle Bezeichner (`x`, `y` und `z`) können an den eingehenden Kanten gebunden werden. Wohlgemerkt ist ‘=’ hier keine Gleichheitsspezifikation sondern ein Prädikat.

Das Netz in Abbildung 8.6 auf der folgenden Seite berechnet den größten gemeinsamen Teiler mit expliziter dynamischer Typprüfung. Prozeduren, die eine Typprüfung vornehmen, können in Guards sehr wohl verwendet werden, wenn alle in den Parametern vorkommenden Bezeichner bei der Auswertung gebunden sind, d. h. wenn kein Pattern-Matching vorgenommen werden muss. Ihre Verwendung mit ungebundenen Bezeichnern ist aber nicht möglich. Das bereits in Abschnitt 7.5.1 (S. 77) diskutierte Beispiel kann also erst auf der nächsten Stufe nach Scheme übertragen werden.

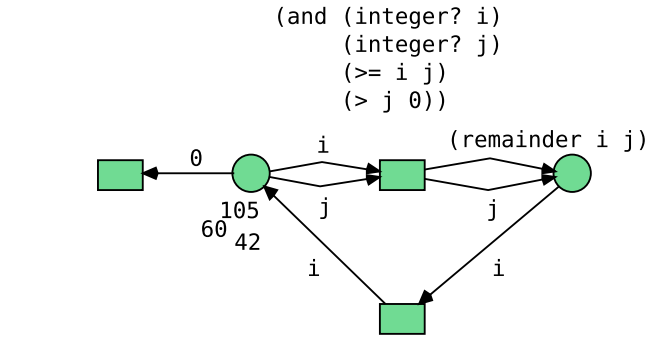

Abbildung 8.6: Das Netz „gcdtyped“

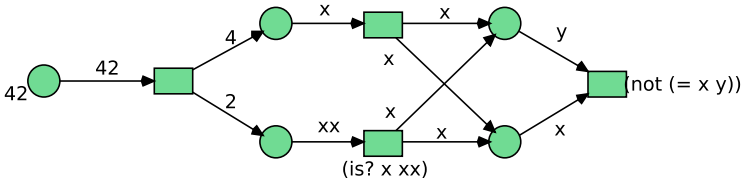

Abbildung 8.7: Das Netz „colored“

8.1.4 Scheme-Coloured-Nets mit Benutzerdefinierten Bindungen

Auf dieser Stufe werden sowohl das Unifikationsprädikat `is?` als auch der Bindungsoperator `bind` eingeführt, damit an Transitionen benutzerdefinierte Bindungen eingeführt werden können. Dies kann zur Vereinfachung der Netzstruktur beitragen, da ansonsten die benötigten Objekte u. U. an Ausgangskanten berechnet und in Stellen gelegt werden müssten.

Das Netz „colored“ in Abbildung 8.7 ist eine direkte (und diesmal richtige) Übersetzung des gleichnamigen Netzes aus [Kummer u. a., 2006b].

In diesem Beispiel könnte statt `(is? x xx)` sowohl `(is? xx x)`

als auch `(bind x xx)` geschrieben werden[2], um die gleiche Bindung zu erhalten. Die Einführung eines neuen Bezeichners ist an dieser Stelle eigentlich gar nicht nützlich. Nützlich wäre z. B. `(bind x (+ xx 1))` um an den ausgehenden Kanten für den Ausdruck `(+ xx 1)` sowohl Platz zu sparen als auch mit der Redundanz die Fehlerquellen zu minimieren.

Die Operatoren zum Einführen benutzerdefinierter Bindungen können als qualitativer Schritt angesehen werden, weil zu ihrer Implementation auf den Zustand der Simulation zugegriffen werden muss (es müssen bei RENEW z. B. lokale Variablen im `VariableMapper` angelegt werden), aber die dazu nötigen Bindungen nicht in der sekundären Umgebung sichtbar werden dürfen. Wenn ihre Implementation gelingt, ist das gleiche Problem für die `action`-Beschriftungen ebenfalls gelöst.

Ab dieser Stufe ist die Reihenfolge der Implementierung nicht mehr zwingend. Es könnten im nächsten Schritt auch schon die Netzexemplare hinzutreten, aber der `action`-Operator gehört noch zur den Scheme-Coloured-Nets und wird daher als nächstes besprochen.

8.1.5 Scheme-Coloured-Nets mit Actions

Mit dem Makro `action` kommt die Möglichkeit hinzu, Nebeneffekte so zu isolieren, dass sie nur einmalig nach der Bindungssuche ausgeführt werden.

In Abbildung 8.8 rechts gegenüberliegend ist unten links ein umfangreicher Deklarationsknoten zu sehen, der die SISC-spezifischen Anweisungen enthält, mit denen Klassen und einzelne Methoden aus Java-Bibliotheken importiert werden. In diesem Fall werden die importierten Bezeichner der globalen Umgebung hinzugefügt und die Übergangsprozedur erhält den Standardwert, wie in Abschnitt 7.3 (S. 70) erklärt.

Dann werden in verschiedenen Aktionen Java-Methoden aufgerufen, die ein einfaches Fenster mit einer Schaltfläche anzeigen und

[2] aber *nicht* `(bind xx x)`

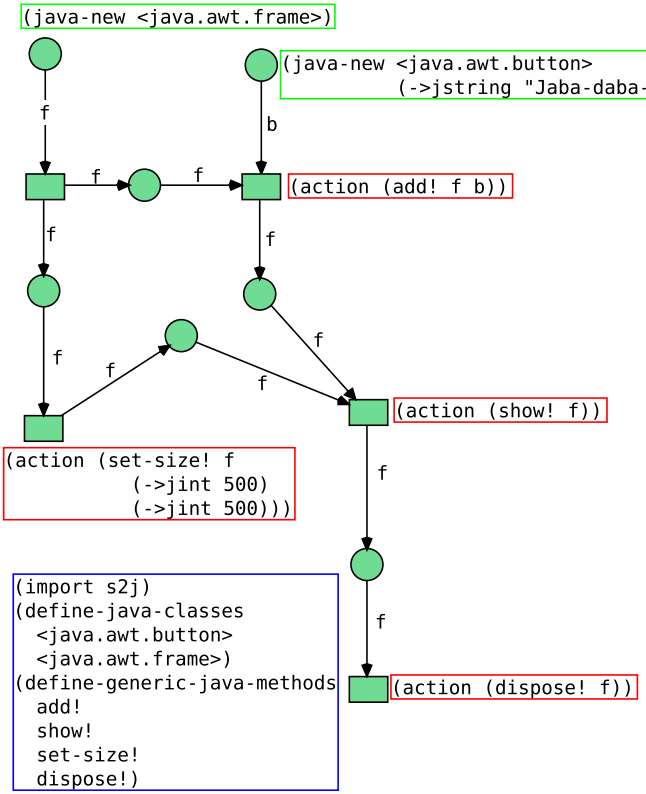

Abbildung 8.8: Das Netz „frame“

wieder vernichten. (Das Netz stammt ebenfalls aus [Kummer u. a., 2006b].)

Durch die Semantik des `action`-Operators soll sicher gestellt werden, dass diese Aktionen erst nach der Unifikation aber vor der Berechnung der Nachfolgemarkierung durchgeführt werden.

An diesem Netz wird auch deutlich, dass SISC vollen Zugriff auf alle Java-Klassen bietet, aber zwischen den Datentypen der beiden Welten sehr streng unterschieden wird. Es gibt keine automatische Konvertierung von Scheme nach Java oder umgekehrt.

Es zeigt sich auch, dass nicht jeglicher Nebeneffekt allein durch den Schutz der globalen Umgebung verhindert werden soll und kann. In jeder Transition wird ein Bezeichner in einer sekundären Umgebung gebunden und der interne Zustand des Objekts wird verändert. Lediglich die Zuweisung an einen der im Deklarationsknoten gebundenen Bezeichner (wie etwa `add!`) wird in diesem Netz schon verhindert. Auch z. B. die Veränderung von Listen mittels `set-car!` oder `set-cdr!` wäre in diesem Netz ohne weiteres möglich.

In Abbildung 8.9 rechts gegenüberliegend sieht man ein Java-Referenznetz aus [Kummer, 2002, S.306], das einen Ausschnitt eines größeren Netzes darstellen könnte, und darunter (Abbildung 8.10) seine Übertragung nach Scheme (bereits mit synchronen Kanälen, also ein Vorgriff auf die übernächste Stufe). Die Problematik bei beiden Netzen besteht darin, dass die Methodenaufrufe von `nameFor` und `idFor` bzw. die Operatoren `name-for-id` und `id-for-name` möglicherweise Nebeneffekte erzeugen, die in eine Aktion gekapselt werden müssten.

Mit Scheme-Referenznetzen lässt sich die Semantik des Java-Referenznetzes *nicht* genau nachbilden. Die Scheme-Prozeduren sind möglichst so zu implementieren, dass sie entfaltbar sind[3], außerdem ist zu empfehlen, dass die Nebeneffekte nicht in den Umgebungen

[3] Also so, dass ein Prozeduraufruf durch sein Ergebnis ersetzt werden kann. Beispielsweise könnten die im Java-Referenznetz verwendeten Methoden in Scheme-Prozeduren gekapselt werden, die mittels `delay` und `force` dafür sorgen, dass die Nebeneffekte nur einmalig auftreten

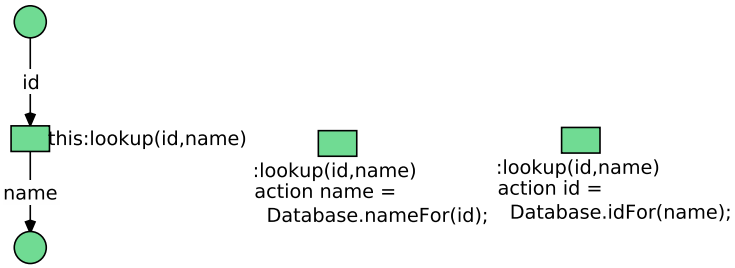

Abbildung 8.9: Bindung mittels Methodenaufruf und Action in einem Java-Referenznetz-System

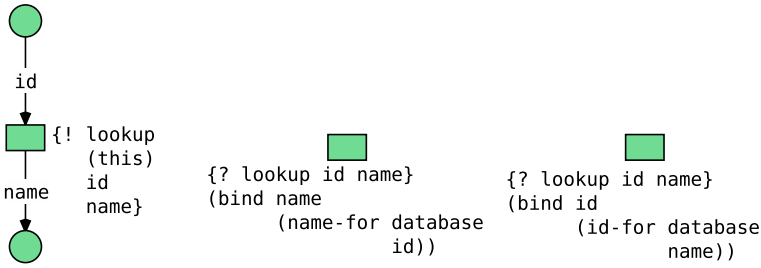

Abbildung 8.10: Scheme-Version des oberen Netzes, ohne Isolierung von Nebeneffekten

sichtbar werden. Die Kapselung von Nebeneffekten mit zusätzlicher Einführung von Bindungen ist in Scheme-Referenzen aber nicht möglich. Sie sollte auch nicht nötig sein, da Scheme ausreichend mächtige Techniken der funktionalen Programmierung anbietet.

Bereits in der vorigen Stufe musste Zugriff auf den Zustand des Simulators genommen werden, neu ist hier lediglich die Anmeldung einer Berechnung beim `CalculationChecker`.

8.1.6 Scheme-Coloured Nets mit manuellen Transitionen

Auf dieser Stufe wird mit der Beschriftung `{manual}` die erste spezielle Auszeichnung hinzugenommen. Im Falle der Implementation

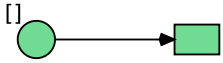

Abbildung 8.11: Das Netz „othernet“

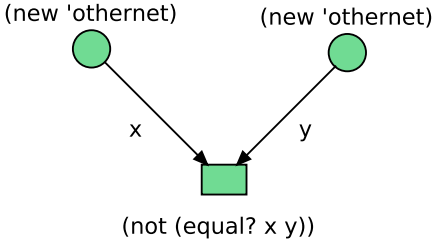

Abbildung 8.12: Das Netz „creator“

als RENEW-Plugin, aber vermutlich auch bei anderen Implementationen, wird hier der Parser für die Netzbeschreibungen etwas modifiziert werden müssen, damit die generierte Struktur (wie etwa Objekte der RENEW-Klasse `de.renew.net.Net`) entsprechend angepasst werden kann.

Diese Technik wird im nächsten Schritt wiederum benötigt.

8.1.7 Scheme-Referenznetze

Für einen Referenznetz-Formalismus werden sowohl Netzexemplare als auch synchrone Kanäle benötigt, dafür müssen sowohl die Operatoren `new` und `this` als auch die Auszeichnungen `?`, und `!` in dieser Stufe hinzu kommen.

Sowohl das Beispiel in Abbildung 8.13, als auch das Netzsystem in den Abbildungen 8.11 und 8.12 stammen aus [Kummer u. a., 2006b]. Im Netzsystem aus `othernet` und `creator` werden lediglich zwei Netzexemplare erzeugt, wobei dies in der initialen Markierung einer Stelle geschieht, was mit Java-Referenznetzen nicht möglich ist. Da es sich beim Operator `new` um eine Prozedur handelt, muss der Name des Netzmusters als Scheme-Ausdruck angegeben werden.

Im Netz `synchro` wird mit `(this)` eine Referenz auf das aktuelle Netzexemplar erzeugt und in einem Downlink verwendet. Außerdem wird eine Transition mit einem Uplink versehen. Die Lokalität

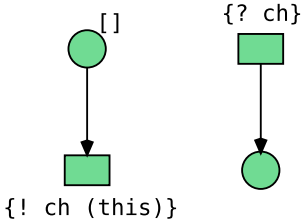

Abbildung 8.13: Das Netz „synchro“

der Transitionen ist direkt an der Beschriftung ablesbar. Beide Transitionen haben die gleiche Lokalität. Der Name des Kanals ist kein Scheme-Ausdruck sondern wird direkt angegeben.

Der wesentliche Schritt auf dieser Stufe ist natürlich die Arbeit mit Netzexemplaren.

Bei der RENEW-Anbindung müssen bei der Übersetzung der Uplinks und Downlinks die entsprechenden Instanzvariablen im erzeugten Netz richtig gesetzt werden. Außerdem muss mit dem Operator `new` eine Netzinstanz erzeugt werden. Dazu muss von dem bereits für Java-Referenznetze implementierten Mechanismus so weit abstrahiert werden, dass Netzexemplare in jeder Beschriftung erzeugt werden können.

Der Operator `this` lässt sich für RENEW sehr einfach implementieren, da das `VariableMapper`-Objekt bei jeder Auswertung eine lokale Variable *`this`* enthält, in der die aktuelle Netzinstanz enthalten ist.

8.2 Weitere Beispiele

8.2.1 Prozeduren als Marken

In Abbildung 8.14 auf der nächsten Seite werden Prozeduren als initiale Markierung erzeugt. Die Prozedur wird auf eine Marke ange-

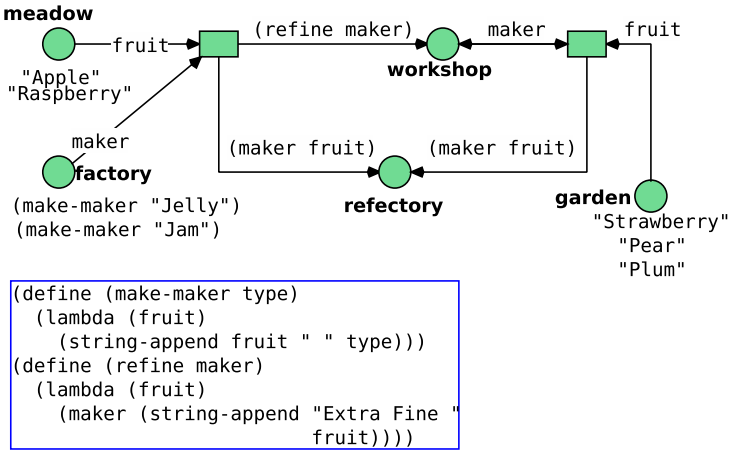

Abbildung 8.14: Das Netz „procedures“

wendet und selbst zur Erzeugung einer neuen Prozedur verwendet.[4] Man sieht an diesem Beispiel auch, dass an Kanten direkt Prozeduren an Bezeichner gebunden werden können, ohne dass `bind` an einer Transition verwendet werden muss.

8.2.2 Continuations als Marken

Auch Continuations können als Marken verwendet werden, was sich zwingend daraus ergibt, dass Continuations Prozeduren sind, die bereits als Marken verwendet werden können.

Es ist aber in der Regel nicht sinnvoll, sie an einer anderen Kante oder Transition zu verwenden, als an der, an der sie eingefangen wurden, zum Einen weil ihre Umgebung für die aktuelle Transition irrelevant ist, zum Anderen weil beim erneuten Aufruf der Continuation der gesamte aufrufende Code wieder ausgeführt wird, wodurch z. B. Teile der Bindungssuche in ihrem alten Kontext erneut

[4] Die Marken des Beispiels sind eine Hommage an die „Schemer“-Bücher von Daniel P. Friedman und anderen, wie z. B. [Friedman und Felleisen, 1996]

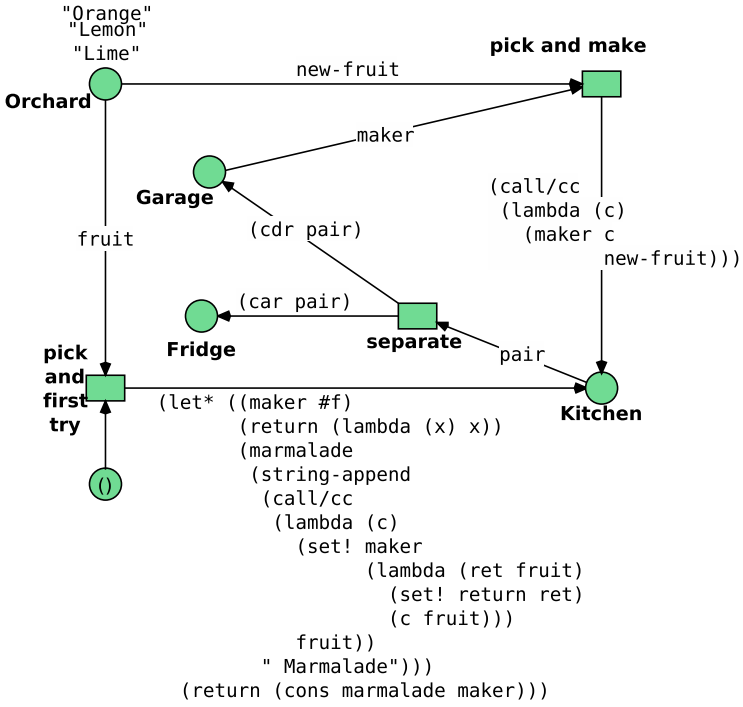

Abbildung 8.15: „Delimited Continuations“ als Marken

durchgeführt werden, und dabei lokale Variablen aktualisiert werden, die für die aktuelle Transition längst nicht mehr relevant sind.

Abhilfe lässt sich schaffen, indem die Continuations durch einen geeigneten Mechanismus limitiert werden. Es gibt eine Menge Aufsätze über „Delimited Continuations“. Einige Publikationen von Oleg Kiselyov zu diesem Thema finden sich im September 2007 unter http://okmij.org/ftp/Computation/Continuations.html und http://okmij.org/ftp/Computation/dynamic-binding.html, vermutlich lassen sich einige der dort verfolgten Ansätze und Implementationen (wie etwa http://okmij.org/ftp/Computation/dynamic-binding.html#DDBinding) direkt zur Verwendung mit SISC und damit auch für Scheme-Referenznetze anpassen.

Das Netz in Abbildung 8.15 auf der vorhergehenden Seite limitiert die im Gültigkeitsbereich der Prozedur `maker` gebundene Continuation – in Abwesenheit eines geeigneten allgemeinen Rahmenwerks – durch explizite Übergabe der aktuellen Continuation. Obwohl es sich um ein triviales Beispiel handelt, in dem ein so mächtiger Kontrollmechanismus gar nicht benötigt würde, ist der `let*`-Ausdruck in dem die Continuation eingefangen wird für eine Kantenbeschriftung unkomfortabel lang. Durch ein geeignetes Rahmenwerk könnten die Anschriften deutlich knapper formuliert werden. Außerdem ist der hier verfolgte Ansatz nur eine Abhilfe für das konkrete Problem und aus verschiedenen Gründen nicht als allgemeine Lösung tauglich.

Das Netz illustriert aber, dass mit oder ohne Rahmenwerk für limitierte Continuations, diese durchaus mit der gebotenen Sorgfalt in Scheme-Referenznetzen verwendet werden können. Es ist also sinnvoll, Continuations generell als Marken zuzulassen.

(Die in diesem Netz verwendete Abhilfe zur Limitierung der Continuation wurde von http://www.double.co.nz/scheme/partial-continuations/partial-continuations.html inspiriert.)

8.2.3 Logikprädikate in Beschriftungen

Das Netz in Abbildung 8.16 auf der nächsten Seite importiert im Deklarationsknoten ein Modul, welches alle im Logiksystem KANREN verfügbaren Bindungen und zusätzlich ein Makro `bind-random-solution` bereitstellt. Das Makro verwendet den Operator `bind` um eine zufällige von `limit` möglichen Lösungen für `x` und `y` zu binden. Im Deklarationsknoten wird die KANREN-Relation `%append` definiert, welche mit dem Makro in einem Guard verwendet wird. (Die Relation wurde aus einem Beispiel in den KANREN-Quellen angepasst.)

Die Bezeichner `x` und `y` werden in der unteren Transition also an zufällige Abschnitte der Liste `z` gebunden.

Die obere Transition verwendet die Scheme-Prozedur *`append`* (nicht zu verwechseln mit der Relation), um zwei Marken wieder zu einer Liste zusammenzufügen.[5]

Das Netz illustriert die Verwendung von `bind` in einem benutzerdefinierten Makro. Das Makro `bind-random-solution` kommt ohne die direkte Hilfe z. B. des `VariableMapper`-Objekts oder anderer Simulator-Objekte aus, die in Beschriftungen auch gar nicht (bzw. nur über sehr unbequeme Umwege) zur Verfügung stehen. Das Makro `bind` hingegen muss auf diese Objekte zurückgreifen.

Die Implementierung eines solchen Makros erfordert unter Umständen ein genaues Verständnis der Bindungssuche, so wie die Implementierung eines Makros in Scheme im allgemeinen auch genaue Kenntnisse der Sprachsemantik erfordert.

Dieses Beispiel illustriert auch die Ausdrucksmöglichkeiten, die sich daraus ergeben, dass beliebige Scheme-Ausdrücke als Beschriftungen zugelassen sind.

[5] Selbstverständlich könnte auch hier die Relation verwendet werden, sogar ohne Verwendung des Makros, da es für den dritten Parameter nur eine Lösung gibt, wenn zwei schon gebunden sind. Die Beschriftung der Ausgangskante könnte also z. B. lauten `(cadar (solution (z)(%append x y z)))`. Zum gleichen Ergebnis führt `(append x y)` und ist außerdem effizienter.

```
(require-extension (lib de/renew/scheme/kanren-nets))
(define limit 20)
;; We can define relations and use them in nets now:
(define %append
  (relation-head-let
   (a b c)
   (any ((relation (l)
                   (to-show '() l l)
                   succeed)
         a b c)
        ((relation (x l1 l2 l3)
                   (to-show `(,x . ,l1) l2 `(,x . ,l3))
                   (%append l1 l2 l3))
         a b c))))
```

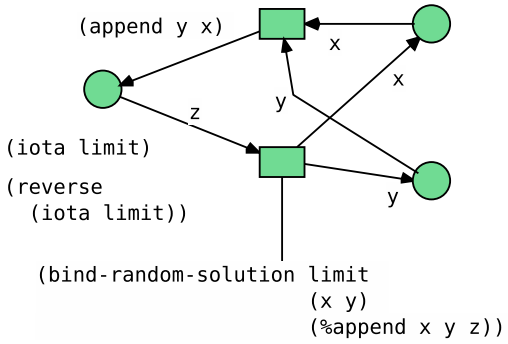

Abbildung 8.16: Logikprädikate in Beschriftungen

9 Bindungssuche

Bei allen konzeptionellen Stufen in Abschnitt 8.1 (S. 95) müssen Bindungen von Transitionen gefunden werden. Im einfachsten Fall (Netze mit Scheme-Objekten in Stellen und an Kanten [S. 96]) ist es bei der RENEW-Integration schon damit getan, dass Scheme-Objekte in Java-Objekte verpackt werden, deren `equals`-Methode dem Scheme-Prädikat `equal?` entspricht.

Schon in der folgenden Stufe ist es aber notwendig, Marken mit Patterns zu vergleichen und ungebundene Bezeichner in den Patterns entsprechend zu binden.

Die in der nächsten Stufe hinzutretenden Guards verlangen lediglich, dass die gefundenen Bezeichner durch Werte ersetzt und die Guard-Ausdrücke ausgewertet und mit `#f` verglichen werden.

Aber sowohl bei den benutzerdefinierten Bindungen als auch bei synchronen Kanälen müssen wieder Bezeichner gebunden werden und in beiden Fällen kann sogar Unifikation erforderlich sein (Wie etwa beim Ausdruck `(is? (cons 'a k)(cons j 'b))` oder im Netz in Abbildung 8.10 (S. 105)).

9.1 Unifikation von S-Expressions

Das Kernstück des Scheme-Formalismus bildet die Unifikation von Scheme-Ausdrücken an Transitionen und Kanten.

Da bereits der in Scheme eingebaute Parser zur Analyse der Ausdrücke verwendet werden soll, liegt es sehr nahe für die Unifikation ebenfalls Scheme, bzw. eine in Scheme implementierte Bibliothek zu verwenden.

Eine Alternative würde darin bestehen, ausschließlich die Unifikationsalgorithmen und -Klassen von RENEW zu verwenden. Da-

für müsste z. B. das Problem gelöst werden, wie etwa der Ausdruck `(cons col type1)` in ein Datenobjekt übersetzt werden kann, das die bereits vorhandenen Unifikationsalgorithmen verstehen. Da solche Probleme bereits von in Scheme implementierten Logiksystemen gelöst werden, erscheint es sinnvoller, auf diese zurückzugreifen.

Ein Vorteil eines in Scheme eingebetteten Logiksystems besteht auch darin, dass es zur Verwendung sowohl in Beschriftungen als auch im Quellcode des Scheme-Plugins leichter von Scheme-Programmierern angepasst werden kann (ähnlich zur Argumentation in 4.4.2 (S. 43)).

Von den für Scheme verfügbaren Logiksystemen kommen SCHELOG ([SCHELOG, 2003]) und KANREN ([KANREN, 2006]) (sowie sein Ableger MINIKANREN ([Friedman u. a., 2005])) in Frage, mit denen sich in den ersten Tests die antizipierten Probleme lösen ließen.

Die Vorteile von SCHELOG und MINIKANREN liegen in ihrer besseren Verständlichkeit, während KANREN ausgereifter zu sein scheint, für gewisse Fälle optimiert ist und mehr Möglichkeiten bietet.

Obwohl alle drei Systeme in der Lage waren, die Testaufgaben zu lösen, war KANREN dennoch als die bessere Wahl, da damit gerechnet wurde, dass während der Implementation Probleme auftreten, die mit den einfacheren Systemen nicht gut lösbar sind.

9.2 Ablauf der Bindungssuche

Bereits vor der Bindungssuche können alle im Netz vorkommenden Scheme-Ausdrücke daraufhin untersucht werden, ob ungebundene Bezeichner in ihnen vorkommen. Dies kann bereits beim Übersetzen des Netzes geschehen, da bei der Simulation zwar neue Bezeichner hinzu kommen (nur sofern `unprotected-environment` verwendet wird) aber in der Regel keine entfallen können. (Außer es wird in einer ungeschützten Umgebung die undokumentierte Pro-

zedur *remprop* verwendet, mit der Bindungen aus der Umgebung entfernt werden können.)

Dazu müssen die Ausdrücke zunächst expandiert werden, so dass keine Makros mehr in ihnen enthalten sind. Durch die Expansion werden die Programme auf einen Sprachkern zurückgeführt, der im Falle von SISC als syntaktische Schlüsselwörter lediglich *begin*, *define*, if, *lambda*, *letrec*, quote, und *set!* enthält.

Von diesen beeinflussen *define*, quote, *lambda* und *letrec* die Bedeutung von in ihren Argumenten vorkommenden Bezeichnern, so dass sie gesondert untersucht werden müssen.

Bindungen an einer Transition können dann wie folgt gesucht werden:

1. Zunächst werden aus den Markierungen der Stellen im Vorbereich der betrachteten Transition mögliche Kandidaten ausgewählt. Die betrachteten Kandidaten müssen, außer für Testkanten, reserviert werden, so dass sie nur für jeweils eine Kante in Erwägung gezogen werden. (Dies entspricht genau der Vorgehensweise des RENEW-Simulators.)

2. Mit einem geeigneten Suchverfahren werden nun sowohl Marken im Vorbereich als auch Guards untersucht.

   Bei allen Auswertungen kann die Kantenbeschriftung verbatim übernommen werden.

   Die Reihenfolge der Untersuchung von Guards und Kantenbeschriftungen ist im Folgenden beliebig.

   - Die Kandidaten können mit dem Pattern an der Kante verglichen werden. Dies geschieht mit Hilfe der Unifikationsbibliothek.
   - Gleichzeitig können bereits Guard-Anschriften ausgewertet werden, sobald alle dafür notwendigen Bezeichner gebunden wurden. Das Makro is? muss dabei gesondert behandelt werden, da vor seiner Auswertung nicht entschieden werden kann, welche Bezeichner möglicherweise schon gebunden werden können.

Beim Fehlschlagen eines Guards kann die Bindungssuche mit den bisherigen Werten abgebrochen und mit den übrigen Kandidaten neu begonnen werden.

- Außerdem können Ausdrücke an Ausgangskanten berechnet werden, sobald alle in ihnen vorkommenden Bezeichner gebunden werden konnten.

3. Sofern alle ungebundenen Bezeichner in allen beteiligten Beschriftungen gebunden werden können, ist eine Bindung für die Transition gefunden und die Transition kann in diesem Modus schalten, falls der Guard-Ausdruck nicht fehlschlägt.

9.3 Bindungssuche bei einer Einbettung in Scheme

Um das obige Verfahren zu konkretisieren und zu präzisieren, kann an dieser Stelle statt Pseudocode schon rein funktionaler Scheme-Code mit KANREN-Operatoren verwendet werden.

Zunächst kann konkretisiert werden, in welchen Ausdrücken ungebundene Bezeichner vorkommen dürfen: KANREN ersetzt die in den Ausdrücken vorhandenen ungebundenen Variablen durch Werte vom Typ `logical-variable` und wertet die Ausdrücke anschließend aus. Dabei können faktisch nur Paare und Listen oder Atome herauskommen, die Logikvariablen und andere Werte enthalten können. Durch dieses Verfahren können aber die Pattern-Listen mit beliebigen rekursiven Funktionen konstruiert werden.

Es wird eine KANREN-Relation benötigt, welche alle Kombinationen von Marken aus einer Stelle bindet. Diese lässt sich als allgemeine Relation formulieren, die alle möglichen Kombinationen der Elemente einer ersten Liste (Kantenbeschriftungen) in einer zweiten Liste (Markierung einer Stelle) enthält.

Diese Relation nenne ich `%reserve`. Sie kann z. B. wie folgt angewendet werden:

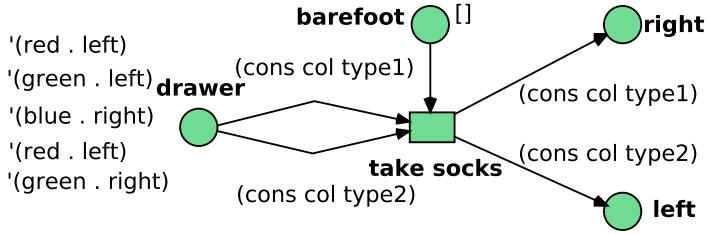

Abbildung 9.1: Wiederholung des Beispiels aus Abbildung 8.3.

```
(solve 1000
       (a b)
       (%reserve (list a b)
                 '(37 93)))
⟹ (((a.0 37) (b.0 93))
    ((a.0 93) (b.0 37)))
```

Das KANREN-Makro `solve` erhält als erstes Argument die Anzahl der maximal gewünschten Lösungen (hier wurde willkürlich 1000 gewählt, normalerweise wird zum Auffinden einer Bindung nur eine Lösung benötigt); das zweite Argument ist die Liste der ungebundenen Bezeichner, danach können mehrere Ziele folgen.

Damit lässt sich aus den ungebundenen Bezeichnern der Ausdrücke und den Kantenbeschriftungen für das Netz in Abbildung 9.1 folgende Anfrage an das Logiksystem formulieren; die Variablen `barefoot` und `drawer` zeigen hierbei auf Scheme-Listen, welche die Marken der gleichnamigen Stellen enthalten:

```
(solve 1000
       (col type1 type2)
       (all (%reserve '(())
                      barefoot)
            (%reserve (list (cons col type1)
                            (cons col type2))
                      drawer)))
```

Ein solcher Ausdruck könnte aus dem Scheme-Programmcode der Netzbeschreibung mit Hilfe des Makrosystems generiert wer-

den.

Die Anfrage liefert als Lösung folgende Substitutionen bzw. Bindungen:

```
(((col.0 red) (type1.0 left) (type2.0 left))
 ((col.0 green) (type1.0 left) (type2.0 right))
 ((col.0 green) (type1.0 right) (type2.0 left))
 ((col.0 red) (type1.0 left) (type2.0 left)))
```

Bemerkenswert am oberen Ausdruck ist, dass die Kantenbeschriftungen direkt in den Ausdruck übernommen werden konnten.

Die Liste der ungebundenen Bezeichner wurde direkt aus der Expansion der Beschriftungsausdrücke extrahiert. Der Operator `all` bildet die logische Konjunktion der Teilziele. In den `%reserve`-Ausdrücken wurden direkt die Kantenbeschriftungen (unbeschriftete Kanten sind implizit mit `'()` beschriftet) und die Markierungen der Stellen (als Variablen) eingesetzt.

Aus den Lösungen lassen sich direkt Bindungen für die ungebundenen Bezeichner extrahieren, die in die Ausdrücke der Ausgangskanten eingesetzt werden können.

Trotz der deklarativen Notation wird KANREN faktisch alle in Frage kommenden Werte durchprobieren, so dass der tatsächliche Ablauf der Unifikation dem im vorigen Absatz beschriebenen sequentiellen Ablauf näher kommt.

Etwas komplizierter wird der Ausdruck in Verbindung mit Guard-Beschriftungen an Transitionen, da Logikvariablen in KANREN nicht in Ausdrücken funktionieren, die eine Typprüfung vornehmen. Nach erfolgreicher Bindung aller *Bezeichner* mit `%reserve` müssen die Logikvariablen darum mit dem Makro `project` reifiziert werden und können im Falle eines Guard-Ausdrucks mit dem Makro `predicate` als weitere Anforderung direkt übernommen werden.

Damit ergibt sich für die Transition mit der Guard-Anschrift (Abbildung 9.2 auf der folgenden Seite) folgender Ausdruck:

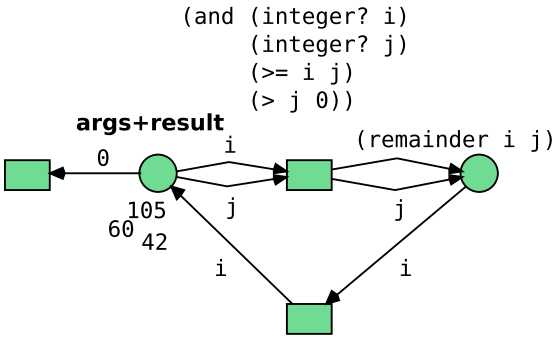

Abbildung 9.2: Wiederholung des Beispiels aus Abbildung 8.6.

```
(solve 20 (i j)
          (all (%reserve (list i j) args+result)
               (project (i j)
                        (predicate (and (integer? i)
                                        (integer? j)
                                        (>= i j)
                                        (> j 0))))))
```

Mit der Lösung:

```
(((i.0 105) (j.0 60))
 ((i.0 105) (j.0 42))
 ((i.0 60) (j.0 42)))
```

Die adäquate Übersetzung von Transitionen mit mehreren Guards stellt eine weitere Schwierigkeit dar, da Guards ja die gesamte Bindungssuche entscheidend beeinflussen und deshalb möglichst früh ausgewertet werden sollten. Gegebenenfalls müssten also Abhängigkeiten zwischen den Ausdrücken berechnet und entsprechend in der Reihenfolge der generierten Teilziele der Anfrage berücksichtigt werden.

Damit ist hier bereits die dritte konzeptionelle Stufe der „Scheme-Coloured-Nets“ (S. 99) angedeutet. Für die nächste Stufe wäre nun zu lösen, wie sich die Operatoren `bind` und `is?` auf diese Weise realisieren lassen.

Mit diesen beiden Ausdrücken lässt sich also schon skizzenhaft ein vielversprechender Ansatz für eine Einbettung von Scheme-Coloured-Nets in die Programmiersprache Scheme erkennen. Er setzt aber voraus, dass die aktuelle Markierung vollständig bekannt ist.

Da die Implementation als RENEW-Plugin verschiedene Vorteile bringt und Scheme-Referenznetze aus RENEW heraus auch in das Format einer Einbettungssprache exportiert werden könnten, sobald sie existiert, wurde die Integration in RENEW als erste Implementation gewählt.

Daher muss sich die Bindungssuche, wenn nicht entscheidende Änderungen am Simulator vorgenommen werden sollen, zunächst an den Ablauf der Simulation in RENEW anpassen. Es zeigt sich, dass diese noch stärker dem in Abschnitt 9.2 (S. 114) beschriebenen sequentiellen Ablauf entspricht. Die Marken werden vom Simulator einzeln mit Kantenbeschriftungen unifiziert, so dass ein kompakter Ausdruck in der oben angegebenen Form nicht ohne weiteres verwendet werden kann.

Teile dieses Ansatzes lassen sich aber durchaus bei der Integration in RENEW verwenden, wie im nächsten Teil gezeigt wird.

Teil III

Umsetzung

10 Verwendete Softwarekomponenten

Abgesehen von den mit SISC und RENEW ausgelieferten Java-Klassen werden keine weiteren Java-Bibliotheken verwendet. SISC selbst bringt einige in Scheme oder Java geschriebene Komponenten, vor allem SRFI-Implementationen mit. Darüber hinaus bilden Teile des Logiksystems KANREN eine wichtige Komponente des Plugins.

10.1 SISC: „Second Interpreter of Scheme Code“

Es wurde bereits beschrieben, dass SISC voll zum R^5RS-Standard konform ist, Java-Integration und erweiterte Möglichkeiten zur Arbeit mit Umgebungen bietet.

Da damit bereits alles über die Eignung der Implementation SISC gesagt wurde, sind nur noch Beschreibungen der weiteren verwendeten Funktionalität sowie auch der Schwächen der Implementation zu hinzuzufügen.

10.1.1 Objektsystem

Da SISC anders als andere Sprachimplementationen für die Java-VM absichtlich keine automatische Konversion von Java-Werten in Scheme-Werte bietet, ist es sinnvoll die verwendeten Java-Klassen so zu kapseln, dass sie in Scheme-gerechterer Weise und auch etwas bequemer verwendet werden können.

Da SISC ein eigenes Objektsystem bietet, liegt es nahe, eine parallele Klassenhierarchie zu den benötigten Java-Klassen aufzubauen.

SISCs Objektsystem ist in drei Komponenten geteilt: Das Typsystem, welches alle Werte (auch Java-Klassen) umfasst, generische Prozeduren, die als Methoden für *alle* (auch primitive) Typen des Typsystems definiert werden können und Klassen, welche multiple Vererbung von generischen Methoden und einfache Vererbung von Feldern bieten.

Das Objektsystem lässt Ähnlichkeiten zu TinyCLOS[1] und damit indirekt zu CLOS[2] erkennen.

Java-Klassen sind zwar Typen des Typsystems, aber keine Klassen im Sinne des Klassensystems von SISC.

Typen sind Werte erster Klasse und können an Variablen gebunden werden, wobei die Bezeichner von Typvariablen konventionell in spitze Klammern gesetzt werden, wie z. B. `<boolean>`.

Diese Konvention wurde auch für die Klassen und Typen des Scheme-Plugins eingehalten.

Mit dem Typsystem und den generischen Prozeduren allein lassen sich bereits Methoden definieren, die dynamisch Java-Klassen zugeordnet werden, dabei lassen sich aber ohne die Möglichkeit Java-Klassen zu beerben keine Felder hinzufügen. Diese Methoden sind auch nur auf der Scheme-Seite verwendbar und sichtbar. Das Objektsystem wird darum dazu verwendet Scheme-Felder zu Java-Klassen hinzuzufügen. Die dazu entwickelte Klassenhierarchie wird in Abschnitt 11.2 (S. 135) beschrieben.

Ein Nachteil dieser Entwurfsentscheidung ist leider, dass der Portierungsaufwand des RENEW-Plugins auf andere Scheme-Implementationen damit ansteigt.

[1] Ursprünglich unter `ftp://ftp.parc.xerox.com/pub/mops/tiny/` verfügbar und mittlerweile von einigen Scheme-Systemen adaptiert und erweitert.

[2] Common Lisp Object System, ein Teil des ANSI Common LISP Standards, siehe u.a. [Graham, 1996]

10.1.2 Verwendete SRFIs

SISC unterstützt eine große Anzahl SRFIs. Verwendet werden im Scheme-Plugin bis jetzt:

[SRFI-1, 1999] Bibliothek zur Arbeit mit Listen. Diese wird von fast allen anderen Scheme-Implementationen unterstützt.

[SRFI-6, 1999] „Basic String Ports“, werden für das Einlesen von Anschriften verwendet.

[SRFI-8, 1999] Ermöglicht bequemes Arbeiten mit multiplen Rückgabewerten.

[SRFI-9, 1999] „Record“ Typen. Das Objektsystem von SISC beruht auf einer Erweiterung von SRFI-9.

[SRFI-11, 2000] Eine weitere Bibliothek für das Arbeiten mit multiplen Rückgabewerten. Der derzeitige Code verwendet zwar nur `receive` aus SRFI-8, aber die Makros aus diesem SRFI sind für komplexere Fälle sinnvoller.

[SRFI-13, 2000] String-Bibliothek, wird ebenfalls zum Parsen von Beschriftungen benötigt, insbesondere für die speziellen Operatoren.

[SRFI-39, 2003] Dynamische Parameter. Mit Hilfe von dynamischen Parametern wird unter anderem die Kommunikation von Beschriftungen wie `is?` oder `this` mit dem RENEW-Simulator realisiert.

In der Regel werden SRFIs von umso mehr Scheme-Implementationen unterstützt, je niedriger ihre Nummer ist. Fast alle der hier verwendeten SRFIs haben überdies eine portable Referenzimplementation; die einzige Ausnahme bildet SRFI-39, da dynamische Parameter am sinnvollsten direkt von der Implementation unterstützt werden sollten.

Die Referenzimplementation von SRFI-39 ist nicht thread-safe und lediglich als Beispiel gedacht; die von SISC unterstützten dynamischen Parameter sind aber thread-safe.

10.1.3 Weitere Funktionalität

Außerdem wird im Scheme-Plugin SISCs Hashtable-Implementation verwendet, welche nicht auf dem entsprechenden SRFI beruht sondern ein andere Schnittstelle anbietet und auf den Java Hash-Tabellen beruht. SISC bietet zwar auf dieser Basis auch eine Implementation von [SRFI-69, 2005], es werden aber Erweiterungen benötigt, die nicht portabel sind.

Als weitere Eigenschaft von SISC wurde bereits das Makrosystem auf Basis von `syntax-case` ([Dybvig, 1992]) erwähnt, welches an einigen Stellen verwendet wird.

Insbesondere prüft das Makro `bind` mit Hilfe eines syntaktischen Guards ob es sich bei seinem ersten Argument um einen Bezeichner handelt. Dies könnte zwar auch zur Laufzeit getestet werden, das aktuell implementierte Makro hat aber den Vorteil, dass es bereits nach dem Erstellen der Anschrift im RENEW-Editor einen Syntaxfehler erzeugt. Syntaktische Guards sind mit den in R^{5}RS definierten *`syntax-rules`*-Transformern nicht möglich.

10.1.4 Schwächen

Da beide Autoren der Implementation SISC in ihrer Freizeit an der Weiterentwicklung arbeiten, gibt es verständlicherweise längere Pausen zwischen der Veröffentlichung neuer Versionen. Die Abfolge der einzelnen Veröffentlichungen war aber bisher schneller als z. B. bei RENEW. Da SISC ein Open-Source Programm ist und der tagesaktuelle Stand der Quellen anders als bei RENEW sogar öffentlich zugänglich ist, wäre es grundsätzlich kein Problem selbst entsprechende Korrekturen zu entwickeln. Der Zeitplan bei der Erstellung dieser Arbeit hat das aber bisher nicht erlaubt. Darum müssen die folgenden beiden Schwachpunkte beim aktuellen Stand der

Implementation in Kauf genommen werden. Beide hier angeführten Schwächen haben die Implementation des Scheme-Plugins auch nicht wesentlich behindert.

Fehlersuche: Schwächen der Implementation SISC (Version 1.16.6) liegen zur Zeit noch im Bereich der Fehlerbehandlung. Im Vergleich zu anderen Scheme-Implementationen sind die Fehlermeldungen oft kryptisch und undurchsichtig. Backtraces sind sind teilweise nicht informativ, insbesondere wenn Funktionen höherer Ordnung oder Callbacks verwendet wurden. Es ist nicht so leicht möglich, lokale Variablen in fehlerhaftem Code zu inspizieren und es gibt nur rudimentäre Funktionalität zur aktiven oder passiven Fehlersuche, die nicht mit dem verwendeten Modulsystem harmoniert.

Aus diesem Grund sollten die Empfehlungen aus dem Handbuch [Miller und Radestock, 2006] dringend eingehalten werden, Module so zu implementieren, dass alle Bindungen auch in die globale Umgebung geladen werden können, was die Fehlersuche in diesen Programmteilen bedeutend erleichtert.

Wenn das Objektsystem von SISC verwendet wird, welches auf Records nach `SRFI`-9 beruht, sind leider die im Backtrace ausgegebenen Werte nicht mehr für Menschen lesbar.

Problematisch ist dies insbesondere bei Implementationen von Java-Interfaces in Scheme, welche von Java-Code aus aufgerufen werden und nur von dort die entsprechende Umgebung erhalten. Die derartig implementierten Methoden können nicht so einfach in der REPL getestet werden.

Leider bleibt in dieser Situation oft nur der letzte Ausweg, Debug-Ausgaben in den Programmcode zu integrieren.

Glücklicherweise ist dies mit Hilfe eines Makros soweit möglich, dass die Argumente des `debug`-Operators nur im Falle eines entsprechend hohen Debug-Levels überhaupt ausgewertet werden.

Weiterhin fehlt eine Möglichkeit des Profilings von Scheme-Code, da das von Java unterstützte Profiling lediglich Informationen über die aufgerufenen Java-Methoden ausgibt, die im Falle von

interpretiertem Scheme-Code immer die selben sind.

Sowohl für die weitere Entwicklung des Scheme-Plugins als auch für die Integration von Fehlerbehandlungsmöglichkeiten in Scheme-Referenznetze wäre es sehr wünschenswert, wenn die Fehlerbehandlungs- und Profiling-Möglichkeiten der Implementation SISC in Zukunft ausgebaut würden.

Für das Profiling und die Ausgabe von Record-Werten gibt es bereits Überlegungen im Fehlerverfolgungssystem bzw. der Mailingliste der SISC-Implementation.

Interaktion von Modulsystem und Umgebungen: SISC 1.16.6 verwendet noch die Version 6.9 des portablen Syntax-Expanders (PSYNTAX) von R. Kent Dybvig und Oscar Waddell (siehe [Dybvig, 1992], erhältlich unter `http://www.cs.indiana.edu/chezscheme/syntax-case/`). Leider wurden offenbar erst in der nächsten Version die Möglichkeiten zur Unterstützung von Umgebungen verbessert.

Vermutlich hat entweder dieses Manko der älteren PSYNTAX-Version, oder ein Fehler in der Behandlung von Umgebungen in SISC die Auswirkung, dass `require-library` und `require-extension` ([SRFI-55, 2005]) nicht in Umgebungen funktionieren, also auch nicht im Deklarationsknoten verwendet werden können, weswegen das Beispiel in Abbildung 8.16 (S. 112) z. Z. nur mit einer modifizierten Version des Scheme-Plugins selbst lauffähig ist, mit dem die benötigte Bibliothek bereits vor dem Parsen der Deklarationsbeschriftungen geladen werden muss.

Damit Scheme-Referenznetze als als Teil einer komplexen Anwendung benutzt werden können ist eine bequeme Möglichkeit zum Import von Bibliotheken unabdingbar.

Mir liegt eine Absichtserklärung des SISC-Erfinders vor, die verwendete PSYNTAX-Version auszutauschen[Miller, 2007].

10.2 KANREN: Logikprogrammierung in Scheme

Mit KANREN (beschrieben in [KANREN, 2006]) liegt ein sehr mächtiges und effizientes System zur Logik-Programmierung vor, welches in portablem R^{5}RS-Scheme geschrieben wurde. Es ist in zwei Varianten verfügbar, zum einen das für Lehrzwecke verwendete MINIKANREN, welches in [Friedman u. a., 2005] eingehend beschrieben wird sowie das hinsichtlich der Performanz und der Eigenschaften des Unifikationsverfahrens optimierte eigentliche KANREN-System. Vorteile von MINIKANREN liegen in der sehr verständlichen Dokumentation und einer deutlich größeren Benutzerfreundlichkeit im Vergleich zu KANREN. Dagegen ist die für KANREN verfügbare Dokumentation zwar sehr umfangreich aber auch sehr technisch. Die Bedienung der beiden Systeme unterscheidet sich leider auch sehr stark, so dass die MINIKANREN-Dokumentation zum Verständnis des komplexeren KANREN-Systems nur eingeschränkt hilfreich ist.

Die Schnittstelle zur Verwendung der Prädikate und Relationen in KANREN beruht fast vollständig auf Makros, was leider den Nachteil hat, dass es nicht verwendet werden kann um dynamische Anfragen aus Beschriftungen zu generieren, da Makros bereits bei der Übersetzung eines Scheme-Programms wirksam werden. Bei einer Einbettung in Scheme wie sie in Abschnitt 9.3 (S. 116) skizziert wurde läge der Fall anders, da die Netze direkt in Scheme-Programme integriert wären und auch gemeinsam mit ihnen übersetzt würden.

Glücklicherweise wird aber gar kein komplettes Logiksystem benötigt. Die Integration in den RENEW-Simulator erfolgt so, dass der RENEW-Simulator einzelne Anschriften miteinander unifiziert um so Bindungen zu erhalten. Es werden also lediglich Terme, Substitutionen, Logikvariablen und Unifikation benötigt. Dieser Teil ist in der Datei `term.scm` der KANREN-Quellen von den Prädikaten und Relationen in `kanren.ss` abgetrennt. Für die Verwendung im

Scheme-Plugin wird also lediglich die Datei `term.scm` eingebunden. Diese stellt mit der Prozedur `unify` und anderen eine prozedurale Schnittstelle zur Termunifikation bereit.

Der KANREN-Quellcode wurde von mir zur bequemeren Verwendung in ein Modul umgearbeitet, wofür größtenteils die Tests ausgelagert und einige wenige Definitionen in den Initialisierungsteil des Moduls bewegt werden mussten.

11 Integration eines Scheme-Interpreters in RENEW

Die erste der zum Schluss des Abschnitts 4.4.4 (S. 53) genannten Anforderungen mit ihren Unterpunkten hat zur Folge, dass zunächst ein Scheme-Interpreter direkt in RENEW integriert werden musste.

11.1 Das Scheme-Plugin

Der Name des Plugins ist „Renew Scheme“ und es wird in den RENEW-Quellen im Unterverzeichnis „`Scheme`“ angelegt.

Wenn nur Dateinamen ohne Pfad angegeben werden, liegen diese in `Scheme/src/de/renew/scheme`.

Die Hauptklasse des Plugins ist
`de.renew.scheme.SchemePlugin`.

Die Klassen der Scheme-Implementation werden in das RENEW-Plugin integriert und es wird über eine bereitgestellte Java-Methode ein Scheme-Interpreter gestartet, der die eigentliche Initialisierung des Plugins übernimmt. Diese erfolgt wie in wie in [Schumacher, 2003] beschrieben. Bereits hier ist Zugriff auf Java-Klassen unbedingt notwendig. Außerdem muss der Scheme-Interpreter in der Lage sein, Module und andere Quelldateien nachzuladen. SISC verwendet hierfür den `ContextClassLoader` des aufrufenden `Threads`.

Für die Konfiguration des Scheme-Plugins werden drei weitere Java-Properties benötigt, die in der Datei `Scheme/etc/plugin.cfg` eingerichtet werden:

```
schemeExtension = de/renew/scheme/renew-scheme
schemeInit = renew-scheme-init!
schemeCleanup = renew-scheme-cleanup!
```

Die Hauptklasse lädt beim Initialisieren das in `schemeExtension` festgelegte SISC-Modul und ruft nach dem Laden die in `schemeInit` festgelegte Prozedur auf. In der `cleanup`-Methode der Hauptklasse wird wiederum die Prozedur in `schemeCleanup` aufgerufen, bevor das Plugin entladen werden kann.

Der gesamte Aufruf zur Initialisierung des Plugins aus Java-Sicht ist also lediglich

```
getAppContext();
eval("(require-extension (lib "
     + _properties.getProperty("schemeExtension") + "))"
     + "(" + _properties.getProperty(("schemeInit")) + ")");
```

Die Methode `eval` der Klasse `SchemePlugin` prüft dabei, ob ein aktueller Scheme-Interpreter bereits läuft und bestimmt danach die Art, wie Scheme-Code von Java aus aufgerufen wird: entweder als „internal“ oder „external call“ (siehe [Miller und Radestock, 2006]).

Die Methode `getAppContext` ruft die entsprechenden Methoden des SISC-API auf, um den Scheme-Interpreter zu initialisieren (vgl. [SISC Javadoc]).

Dieses Prinzip lässt sich auch für spätere, in Scheme geschriebene RENEW-Plugins anwenden. Diese sollten durch Erweiterung der `SchemePlugin` Klasse bequem deren `init` und `cleanup` Methoden erben können, so dass weitere Scheme-Plugins sich auf noch weniger Zeilen Java beschränken können.

Noch einfacher wäre die Einbindung von weiteren Scheme-Plugins allein über die Properties. Es wäre also denkbar, dass das Scheme-Plugin nach dem Start nach weiteren von ihm abgeleiteten Plugins sucht, deren Module lädt und ihre Init-Prozeduren aufruft. Das Scheme-Plugin würde somit als eine Art Scheme-spezifischer Plugin-Mechanismus (ähnlich der RENEW-Klasse `PluginLoader`) fungieren. Diese Plugins könnten dann ausschließlich in Scheme implementiert werden.

Außerdem lässt sich der Mechanismus dazu nutzen, während der Entwicklung experimentelle Programmteile, automatische Tests oder andere Abwandlungen zu laden.

Bei der Initialisierung des Plugins werden der Formalismus und der Compiler für Scheme-Referenznetze, sowie Kommandos und Menü-Einträge als Java-Objekte erzeugt und an zentralen Objekten der RENEW Umgebung angemeldet. Dazu müssen in Scheme teilweise Interfaces implementiert, teilweise Java-Klassen erweitert werden.

Da SISC noch nicht die Erweiterung von Java-Klassen direkt unterstützt, sind neben der Java-Klasse, die den Scheme-Interpreter startet, noch Adapterklassen notwendig, die in ihrem Konstruktor ein in SISC implementiertes Java-Interface als Argument bekommen. Die Adapterklasse erbt von der zu erweiternden Java-Klasse, deren Methoden in Scheme implementiert werden. An vielen Stellen wurde aber in RENEW schon mit Java-Interfaces gearbeitet und einige benötigte Adapterklassen stehen bereits zur Verfügung.

Außerdem muss für jeden Thread des `renew`-Prozesses in SISC eine neue dynamische Umgebung (Klasse `DynamicEnvironment` im Paket `sisc.env`) und ein neuer Interpreter (Klasse `Interpreter` in `sisc.interpreter`) erzeugt werden, da diese nicht über mehrere Threads hinweg verwendet werden dürfen (siehe [SISC Javadoc] und [Miller und Radestock, 2006, Kapitel 8]). Diese müssen von Java aus erzeugt werden. Sinnvollerweise sollte eine Java-Methode zur Verfügung stehen, welche die korrekte Behandlung von dynamischer Umgebung und Interpreter sicher stellt. Die dazu bereit gestellte `eval` Methode der `SchemePlugin` Klasse ist schon etwas umfangreicher als der eben gezeigte Code, da geprüft werden muss, ob es aktuell einen gültigen Interpreter gibt.

Bei einem einzelnen Scheme-Plugin ist dieser Verwaltungsaufwand aber unnötig, weil er für die in Scheme implementierten Java-Interfaces bereits automatisch von der SISC-Laufzeitumgebung erledigt wird. Wenn ein weiteres Scheme-Plugin in einem neuen Thread laufen soll, wird dies aber relevant.

Die interaktive REPL wird in der Datei `scheme-commands.scm` als Kommando für den RENEW-Prompt implementiert (Klasse `CLCommand` im Paket `de.renew.plugin.command`), so dass nach der Eingabe des Kommandos „`scheme`“ wie in einem üblichen Scheme-Interpreter gearbeitet werden kann. Dies erlaubt die Integration eines laufenden RENEW Prozesses mit Scheme-Plugin in gängige Entwicklungsumgebungen für Scheme, wie z. B. EMACS.

Eine denkbare Erweiterung dieses Konzepts wäre die Möglichkeit, im Falle eines Fehlers oder bei einem Breakpoint an einer Transition, eine REPL aufzurufen, die im entsprechenden Stackframe in der sekundären Umgebung der Transition situiert ist, so dass der Stack und Objekte zur Fehlersuche interaktiv inspiziert werden können und die aktuelle Continuation aufgerufen bzw. angepasst werden kann. Dies wird aber erst komfortabel realisierbar, sobald die momentan nur sehr rudimentären Debugging-Möglichkeiten von SISC in einer späteren Version bedeutend erweitert wurden.

Damit bei der Entwicklung des Scheme-Plugins die Änderungen an den Scheme-Programmteilen zur Laufzeit aktiv werden können, ist noch ein Befehls-Skript für das Betriebssystem nötig, welches das fertig übersetzte Scheme-Plugin aus dem Plugin-Pfad entfernt und den `ClassPath` beim Aufruf der virtuellen Maschine auf das Quellverzeichnis setzt. Statt des Moduls `de/renew/scheme/renew-scheme` wird vom Skript `de/renew/scheme/renew-scheme-debug` geladen, welches alle Definitionen in der globalen Umgebung sichtbar macht und zusätzlich die experimentelle Profiling-Bibliothek lädt, so dass Bindungen (sogar während der laufenden Simulation) einfach in der REPL neu definiert, mit Breakpoints versehen oder anderweitig instrumentiert werden können.

Bei kleineren Korrekturen am Quellcode muss damit auch nicht das gesamte Plugin neu geladen werden. Da der SISC-Compiler (bis Version 1.16.6) sehr langsam arbeitet, ist diese Abkürzung auch dringend erforderlich.

Dieses Script ist bisher nur als Shellscript für Unix-ähnliche Systeme implementiert (in `Scheme/bin/sisc-renew-debug.sh`). In

Abwandlung des `renew` Befehlsscripts wird die Java-VM mit dem passenden `Classpath` aufgerufen und dann über das Kommando `script` erst die GUI und dann die Scheme-REPL mit dem Kommando `scheme` gestartet.

Wird das Script unter einem Namen aufgerufen, der die Zeichenfolge „`emacs`“ enthält (z. B. über eine Kopie oder einen symbolischen Link namens `emacs-sisc-renew-debug.sh`), werden einige Konfigurationsoptionen anders gesetzt, so dass die REPL besser in einem „Inferior Scheme Buffer“ im Emacs integriert werden kann. Durch das Zusammenspiel der Modi `inferior-scheme-mode` und `compilation-shell-minor-mode` lässt sich Emacs bequem als Entwicklungsumgebung für in RENEW laufenden Scheme-Code verwenden. Veränderungen am Quellcode des Scheme-Plugins lassen sich dann z. B. direkt aus dem entsprechenden Emacs-Buffer an die REPL senden.

Die Scheme-Teile des Plugins wurden ausschließlich so entwickelt. Für die Java-Klassen wurde aber größtenteils die Entwicklungsumgebung ECLIPSE[1] verwendet.

11.2 Verwendung der RENEW-Klassen, Interfaces und Objekte in Scheme

Java-Interfaces können in SISC mit Hilfe sogenannter „Java-Proxies“ implementiert werden, die ein Scheme-Interface für dynamische Proxy-Klassen (`java.lang.reflect.Proxy`) zur Verfügung stellen. Der SISC-Operator `define-java-proxy` erhält einen Namen, eine Parameterliste, eine Liste von Java-Interfaces und für jede implementierte Methode eine Scheme-Prozedur, an die der Methodenaufruf weitergereicht wird. Die Verwendung von `define-java-proxy` erzeugt einen Scheme-Operator des angegebenen Namens, der beim Aufruf ein Objekt erzeugt, dass die angegebenen Java-Interfaces implementiert.

[1] `http://www.eclipse.org/`

Mit Hilfe dieser Proxies kann mit Teilen der RENEW-API kommuniziert werden, aber da sie keine Instanzenvariablen haben können, wird ein Verfahren benötigt, mit dem zusätzliche Information zu den Proxy-Objekten gespeichert werden kann. Dazu wird eine SISC-Klassenhierarchie aufgebaut, deren Objekte jeweils ein Proxy-Objekt mit Zusatzinformationen speichern.

Da die Implementationen und Interface-Listen der Proxies oft wiederverwendet werden müssen, wenn mehrere Interfaces implementiert werden sollen, werden die Proxy-Konstruktoren selbst in eine Klassenhierarchie eingegliedert und mit Hilfe geeigneter Methoden werden diese Informationen nur zur jeweils übergeordneten Klasse ergänzt. Es ergeben sich also 3 parallele Hierarchien:

1. Die Hierarchie der Java-Klassen und Interfaces
2. Proxies: Die Hierarchie der Scheme-Klassen von Proxy-Implementationen
3. Wrapper: Die Hierarchie der Scheme-Klassen von Proxies mit Zusatzinformationen

An der Wurzel der Wrapper-Hierarchie auf Scheme-Seite steht die Klasse `<java-wrapper>`. Diese implementiert das dafür geschaffene Interface `de.renew.scheme.SchemeWrappable`, dessen einzige Methode `getSchemeWrapper` das Wrapper-Objekt zu einem Proxy-Objekt liefert.

Einige Klassen werden direkt von `<java-wrapper>` abgeleitet, da die in ihnen benötigten Interfaces nicht miteinander vermischt werden müssen.

Für die bequemere Vermischung und Vererbung von Interface-Implementationen wird die Klasse `<java-proxy>` von Scheme-Wrapper abgeleitet (In der Datei `oo-wrappers.scm`). Jede von ihr abgeleitete Klasse muss die drei Methoden `proxy-interfaces`, `proxy-dispatcher-names` und `proxy-converter-procs` implementieren, welche die Liste ihrer implementierten Interfaces, Methodennamen und Konvertierungsprozeduren zu denen der beerbten

Unterklasse von `<java-proxy>` hinzufügen. Das Ergebnis der Methodenaufrufe für die gesamte Klassenhierarchie wird dann in der `initialize`-Methode der Klasse `<java-proxy>` exemplarisch für die Klasse gespeichert, so dass die Listen nicht für jedes Objekt neu erzeugt werden müssen. Für die Unterklassen von `<java-proxy>` muss daher keine eigene `initialize`-Methode implementiert werden.

Daraus ergibt sich mit sehr wenigen und kurzen Klassen- und Methodendefinitionen eine komplexe Hierarchie von Proxy-Klassen, die völlig dynamisch generiert wird.

Dieses System wurde vor allem wegen seiner hohen Flexibilität gewählt, da erstens erwartet wird, dass in näherer Zukunft direkt Java-Klassen in SISC beerbt werden können und dieser Teil der Implementation dann komplett ausgetauscht werden kann und zweitens die Klassenhierarchie während der Experimentierphasen oft vollständig neu strukturiert werden musste (vgl. Kapitel 12 (S. 139) und Abschnitt 14.2.6 (S. 196)).

Die Wrapper-Klassen haben ihrerseits eine etwas flachere Hierarchie. In der Regel wurde eine abstrakte Wurzelklasse implementiert und die verwendeten Klassen direkt davon abgeleitet.

Die komplette Hierarchie mit den Aggregationsbeziehungen der Proxy- und Wrapper-Klassen ist in vereinfachter UML-Notation in Abbildung 12.1 (S. 145) zu sehen, die implementierten Java-Interfaces wurden weggelassen, lassen sich aber zum Teil an den Namen der Proxy-Klassen erkennen.

12 Integration in den RENEW-Simulator

Obwohl RENEW ein Open-Source Projekt ist, unterscheidet es sich von einigen besser bekannten, größeren Open-Source Projekten in den folgenden Punkten:

- Teile der Dokumentation sind nicht öffentlich zugänglich.
- Gleiches gilt für das Bugtracking-System und das CVS-Repository. Die öffentlich zugänglichen Quellen geben jeweils die aktuelle Release-Version wieder.
- Die wesentliche Dokumentation der Implementation [Kummer u. a., 2006a], entspricht wie auch die Anmerkungen in [Kummer, 2002] noch RENEW-Version 1.6. Entscheidende Änderungen der Architektur lassen sich in [Schumacher, 2003] nachlesen, entsprechen aber auch nicht mehr dem allerneusten Stand. Damit gibt es keine aktuelle Dokumentation zur Implementation.
- Der Quellcode des Simulators selbst ist außerdem, im Gegensatz zu anderen Programmteilen, nur spärlich dokumentiert. (An den Konzepten des Simulators hat sich aber wiederum seit längerem kaum etwas geändert.)

Im Grunde ist es nur dann realistisch möglich, in die Entwicklung von RENEW einzusteigen, wenn man die Chance hat an einem entsprechenden Projekt in der Hamburger Informatik teilzunehmen und dort Kontakt zu den Hauptentwicklern zu knüpfen. RENEW ist

damit in seinem Entwicklungsprozess deutlich weniger offen als viele andere Open-Source-Projekte.

Hauptsächlich auf Grund der etwas spärlichen Dokumentation war schon bei der Konzeption dieser Arbeit klar, dass die Integration in den RENEW-Simulator unter Umständen sehr viel Zeit in Anspruch nehmen und sogar scheitern könnte.

Da die Dokumentation in [Kummer u. a., 2006a] und [Kummer, 2002] oft eher den Entwurf als die genaue Schnittstelle zur Integration eines neuen Formalismus beschreibt, konnte die tatsächliche Funktionsweise des Simulators oft besser aus dem Quellcode und eigenen Experimenten erschlossen werden.

Die folgenden Abschnitte beschreiben die praktischen Schritte, die zur Integration eines neuen Netzformalismus nötig sind, auch wenn dieser eine völlig andere Anschriftsprache als Java verwendet.

Dabei ist die Kommunikation mit dem RENEW-Simulator bei weitem der schwierigste und umfangreichste Teil der Implementation. Nachdem dieser Teil fertig gestellt war, sind Erweiterungen wie Netzexemplare und flexible Kanten mit erstaunlich wenig Aufwand gelungen, was sich auch im Umfang der entsprechenden Absätze dieses Kapitels niederschlägt.

12.1 Der Ansatz der Implementation

Bei der Integration von Scheme-Referenznetzen in den RENEW-Simulator wurde ein möglichst sparsamer Ansatz verfolgt. Bevor überhaupt die erste Zeile Code geschrieben war, wurde nach der Schnittstelle gesucht, die mit möglichst wenig Modifikationen am bisherigen RENEW-Code und möglichst wenig Duplikation von Funktionalität zum gewünschten Ergebnis führt.

Der gewählte Ansatz beschränkt sich im wesentlichen darauf, Java-Interfaces und Klassen aus den Paketen `de.renew.unify` und `de.renew.expression` zu verwenden bzw. zu implementieren.

Möglicherweise hätte ein etwas dichter am Simulator liegender Ansatz ebenso zum Erfolg geführt, in dem für Scheme-Referenznet-

ze eigene Netzelemente in `de.renew.shadow` und dazu passende `Occurence`- und `Binder`-Klassen eingeführt worden wären. Damit hätte sich unter Umständen das in Abschnitt 9.3 (S. 116) beschriebene Verfahren direkt durchführen lassen. Auf jeden Fall wäre die damit erreichte Flexibilität vermutlich höher. Es gibt aber in diesen Bereichen des RENEW-Klassenbaums wenige geeignete Java-Interfaces, so dass zu diesem Zweck viel mehr Adapter- oder abgeleitete Klassen notwendig gewesen wären. Vermutlich lassen sich mit einem derartigen Ansatz zwar tiefgreifendere Modifikationen am Simulator vornehmen, aber zum Preis eines höheren Aufwands für die Implementation und einiger Duplizierung von Funktionalität. Beim Ansatz über die `Expression`-Implementationen kommen immerhin nur die bereits vorhandenen `Binder`- und `Occurence`-Objekte des RENEW-Simulators zum Einsatz. Bei einem tieferen Ansetzen hätten diese aber erweitert oder neu implementiert werden müssen.

Tiefgreifende Änderungen waren aber offensichtlich nicht notwendig. Der verfolgte Ansatz hat zum Erfolg geführt und kommuniziert im wesentlichen über das Interface `de.renew.expression.Expression` mit dem Simulator.

12.2 Übersetzen von Scheme-Referenznetzen

Nach der Integration des Scheme-Interpreters und der REPL wurde als nächstes der Formalismus und Compiler für Scheme-Beschriftungen angegangen.

In der Datei `formalisms.scm` wird zunächst direkt mittels `define-java-proxy` ein Operator zum Erzeugen von Formalismus-Implementationen definiert. Diese implementieren das Interface `SingleNetCompilerImplementor`, welches an den Konstruktor von `SingleNetCompilerAdapter` weitergegeben wird. (Beide liegen im Paket `de.renew.formalism.base` unterhalb des Verzeichnisses `Scheme`; sollten sich diese Klassen als verwendbar für weitere Beschriftungssprachen erweisen, wäre es sinnvoll sie in das `Formalism`-Plugin zu verlegen.) Dieser Adapter erbt von

`AbstractSingleNetCompiler` und ruft die Methoden des übergebenen `SingleNetCompilerImplementor`-Objekts auf.

Es wäre genau so gut möglich gewesen, das Interface `ShadowCompiler` direkt zu implementieren, aber da einige Aufgaben bereits von `AbstractSingleNetCompiler` allgemein gelöst werden, hätte dies zu einiger Redundanz geführt. Die RENEW-Klasse `AbstractSingleNetCompiler` ist eine abstrakte Klasse, von der die Compiler für die meisten Java-Formalismen direkt erben. Dass sich die Methoden dieser Klasse direkt auf Scheme-Referenznetze anwenden lassen ist ein schönes Zeugnis für die Wiederverwendbarkeit und Abstraktion des bisherigen RENEW-Codes.

Das hier angewandte Verfahren lässt sich nutzen, um Java-Klassen mit in SISC implementierten Methoden zu erweitern, so lange SISC-Code noch nicht direkt Unterklassen von Java-Klassen bilden kann.

In `scheme-formalisms.scm` werden Prozeduren und Methoden zum Einlesen und Prüfen von Beschriftungen sowie zur Fehlerbehandlung hinzugefügt und eine Klasse `<scheme-compiler>` gebildet, die den konkreten Formalismus implementiert.

Die Methoden zum Übersetzen und Prüfen von Deklarationsknoten, Stellen, Kanten und Transitionen wurden mit ihren Helferprozeduren in einzelne Dateien mit dem Präfix „`scheme-formalism-`“ ausgelagert.

Grundsätzlich werden alle Anschriften mit der Prozedur *`read`* gelesen, wobei eine Überprüfung der S-Expression-Syntax erfolgt, und dann probeweise expandiert, wobei die Makro-Syntax (also auch die der Operatoren `bind` und `is?`) überprüft wird. Außer beim Deklarationsknoten wird dann für jeden Ausdruck ein Objekt erzeugt, welches das Interface `de.renew.expression.Expression` implementiert, und dem entsprechenden Netzelement hinzugefügt.

Diese Objekte werden in Abschnitt 12.3 auf der folgenden Seite genauer beschrieben.

Aus dem Deklarationsknoten wird nach dem in Abschnitt 7.3.2 (S. 71) bereits spezifierten Verfahren die globale Umgebung und die Übergangsprozedur bestimmt und für den jeweiligen Compiler

und das Netzmuster gespeichert.

Für Stellen wird ein Objekt vom Typ `ExpressionTokenSource` erzeugt, Transitionen und Kanten erhalten direkt Objekte der Klasse `Expression`, bzw. im Falle von flexiblen Kanten auch `Function`, die vom Scheme-Formalismus aber nicht benötigt werden.

Die Besonderheiten beim Parsen der speziellen Operatoren an Transitionen werden in Abschnitt 12.8 (S. 169) erklärt, flexible Kanten erfahren eine Sonderbehandlung, die in Abschnitt 12.3 erklärt wird.

12.3 Scheme-Werte als Marken in RENEW

Ausdrücke an Netzelementen werden durch Objekte der Klasse `<unifiable-expression>` implementiert. Bei der Auswertung werden Objekte der Klasse `<unifiable-s-expression>` erzeugt, die Scheme-Ausdrücke darstellen und konkrete Werte der Klassen `<unifiable-value>` bzw. `<collection-value>` erhalten können.

Das bei der Übersetzung des Deklarationsknotens erzeugte `<environment+transformer>`-Objekt wird in einem Hash unter dem Namen des Netzmusters im `<shadow-compiler>`-Objekt gespeichert, von dem es pro Simulation jeweils nur eins gibt.

Bei der Übersetzung der übrigen Netzelemente werden für alle Scheme-Ausdrücke Objekte vom Typ `<unifiable-expression>` erzeugt. Diese enthalten ein Proxy-Objekt vom Typ `<expression-proxy>` welches das Interface `de.renew.expression.Expression` direkt implementiert und die Implementation des Interfaces `de.renew.scheme.SchemeValue` von der Proxy-Klasse `<abstract-value-proxy>` teilweise erbt und teilweise überschreibt.

Das Interface `de.renew.scheme.SchemeValue` überschreibt die Methoden `equals`, `hashCode` und `toString` der Java-Klasse `Object`. Dies wird insbesondere für die von `<unifiable-value>` abstammenden Klassen benötigt, da deren Instanzen direkt als Markenobjekte im Simulator verwendet werden. Die Methoden `equals`

und `hashCode` werden so implementiert, dass sie mit der Scheme-Prozedur `equal?` konsistent sind, wodurch Listen als Markenobjekte verwendet werden können (siehe auch Abschnitt 8.1.1 (S. 96)). Eine konsistente Implementation der Methode `hashCode` ist dabei wichtig, da Markenobjekte vom Simulator auch in Hash-Tabellen abgelegt werden.

Die Methode `toString` wird für die Markenobjekte so implementiert, dass deren Anzeige in der grafischen Simulation der Ausgabe der Scheme-Prozedur *`write`* entspricht, vor allem damit Strings im Simulator als solche zu erkennen sind.

Die `<unifiable-expression>`-Instanzen werden entweder direkt an die Netzelemente angefügt (wie bei Transitionen), oder es werden erst Objekte vom Typ `<arc>` bzw. `<flexible-arc>` für Kanten oder flexible Kanten, bzw. des Typs `ExpressionTokenSource` für Stellen erzeugt.

Abbildung 12.1 rechts gegenüberliegend zeigt die gesamte Klassenhierarchie der Proxy- und Wrapper-Klassen. Oben an der Wurzel des Baums ist `<java-wrapper>` die höchste Abstraktionsstufe, während an oder nahe bei den Blättern des Baums die Klassen zu sehen sind, die instanziiert und als Beschriftungen bzw. Markenobjekte verwendet werden.

Gepunktet umrandete Klassen werden niemals instanziiert sondern stellen lediglich Abstraktionsstufen bzw. Teilimplementationen bereit. Gestrichelt umrandete Klassen werden als Proxies in Wrapper-Objekten verwendet bzw. bilden Abstraktionsstufen in der Proxy-Hierarchie.

Fett umrandete Klassen werden tatsächlich instanziiert.

Die unterschiedlichen Klassen kommen dabei in unterschiedlichen Phasen der Simulation zum Einsatz. `<arc>`, `<flexible-arc>` sowie `<unifiable-expression>` werden direkt vom Compiler erzeugt und erhalten eine Referenz auf den aktuellen Compiler in einem Feld. Direkt bei ihrer Erzeugung werden die `<unifiable-expression>`-Objekte analysiert, wobei ein Objekt vom Typ `<expression-analysis>` herauskommt, welches bei der Auswertung wiederverwendet wird. Das Analyseverfahren erklärt

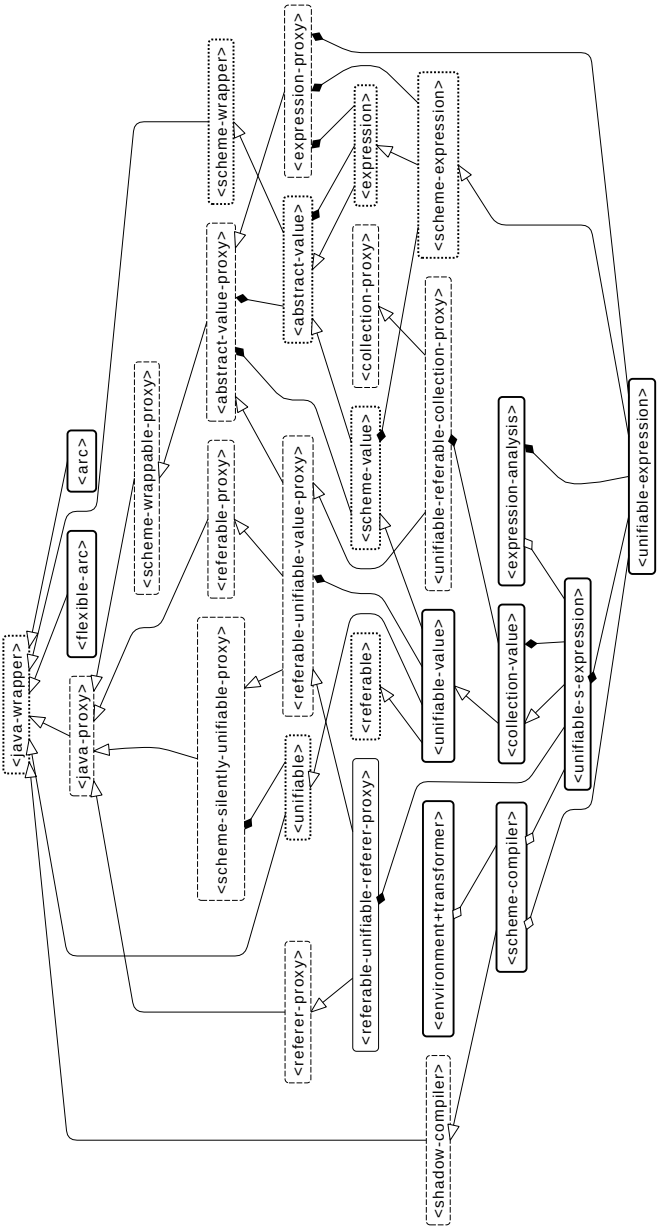

Abbildung 12.1: Hierarchie der Scheme-Wrapper- und Proxy-Klassen für Java-Objekte

Abschnitt 12.4 („Analyse von Ausdrücken“, S.146).

Die erste aufgerufene Methode des `Expression`-Interfaces ist häufig `startEvaluation` wobei Ausdrücke ohne ungebundene Bezeichner direkt ein Objekt vom Typ `<collection-value>` (siehe Abschnitt 12.6 („Flexible Kanten“, S.166)) bzw. `<unifiable-value>` an den Simulator zurückliefern. Stellenbeschriftungen *müssen* ein solches Objekt liefern.

Falls der Ausdruck nicht direkt ausgewertet werden kann, kommen die in den Abschnitten 12.5.1 („Unifikation von Termen“, S.152) und 12.5.2 („Backlinks und Listeners“, S.155) beschriebenen Verfahren zum Einsatz, wobei unter anderem Objekte vom Typ `<unifiable-s-expression>` erzeugt und in späteren Phasen der Simulation verwendet werden.

Die Methode `startEvaluation` wird also in der Initialisierungsphase des Netzes einmal für jede Stellenbeschriftung aufgerufen und bei jeder Prüfung einer Transition. Selbst bei Ausdrücken ohne ungebundene Variablen kann aber der Rückgabewert nicht abgespeichert werden, da es sein kann, dass der entsprechende Ausdruck nicht definit ist (wie etwa bei Ein- und Ausgabeoperationen, Zufallsgeneratoren oder allgemeiner bei Closures mit veränderlichem Zustand oder anderem nebeneffektbehaftetem Code).

12.4 Analyse von Ausdrücken

Bei der Analyse des Ausdrucks eines `<unifiable-expression>`-Objekts wird ein Objekt vom Typ `<expression-analysis>` erzeugt (beide in der Datei `unifiable-expression.scm`).

In ihm werden die Eigenschaften eines Ausdrucks gespeichert, die unabhängig von der sekundären Umgebung sind. Die meisten dieser Eigenschaften beruhen auf der Menge der freien Bezeichner im Ausdruck. Deswegen wird die Untersuchung von Ausdrücken auf ungebundene Bezeichner in einem weiteren Unterabschnitt genauer erklärt.

12.4.1 Das Analyse-Objekt

Ein Objekt vom Typ `<expression-analysis>` hat folgende Felder:

`expansion` Dieses Feld enthält das Ergebnis der Makro-Expansion des Ausdrucks in einer frischen sekundären Umgebung. Auf Grund der in Kapitel 6 (S. 59) vorgenommenen Einschränkung der erlaubten Übergangsfunktionen kann diese Expansion einmalig beim Übersetzen des Netzes geschehen. Zusätzlich zum Makroexpander `sc-expand` wird hier nach der Analyse die Prozedur `optimize` angewendet, die günstigenfalls bereits Optimierungen vornimmt.

Die Optimierung darf aber erst nach der Analyse vorgenommen werden, da bei der Optimierung u. U. ungebundene Bezeichner aus dem Ausdruck entfernt werden, sofern klar ist, dass sie nichts zum Wert des Ausdrucks beitragen.

Sofern beim Simulationslauf *`eval`* aufgerufen werden muss, kann statt des Ausdrucks selbst die Expansion evaluiert werden, was zwei Compilerphasen einspart.

`unbounds` Die Expansion wird in einer frischen sekundären Umgebung mit Hilfe der Prozedur `unbound-symbols/expanded` auf ungebundene Bezeichner untersucht, die in diesem Feld gespeichert werden. Das Verfahren wird in Unterabschnitt 12.4.2 (S. 149) näher erklärt.

`evaluable` Sofern das Feld `unbounds` leer ist, ist dieses Feld `#t` und der Ausdruck kann direkt einen Wert liefern, sonst `#f`.

`logical-variables` Aus der Liste der `unbounds` wird eine Liste von Objekten des in KANREN definierten Record-Typs `logical-variable` erzeugt. Dabei ist es für die Unifikation wichtig, dass identische Bezeichner logische Variablen erhalten, die zueinander `eq?` sind. Der Einfachheit halber werden deshalb die logischen Variablen für das gesamte Netzmuster in einem Cache aufbewahrt. Dies ist keine zu grobe

Vereinfachung, da es nicht auftreten kann, dass die Unifikationsergebnisse einer vollkommen anderen Transition Auswirkungen auf die aktuelle Transition haben, weil die partiellen Bindungen bzw. die Zwischenergebnisse der Unifikation wiederum für jede Transitionsinstanz (sogar für jeden beteiligten Ausdruck vom Typ `<unifiable-s-expression>`) einzeln gespeichert werden. Eine logische Variable in der Implementation durch KANREN hat keinen Wert oder Zustand, sie ist lediglich ein Platzhalter für einen Wert, der durch eine Substitution bestimmt wird.

`logical-term` Die ungebundenen Bezeichner werden in einer frischen sekundären Umgebung probeweise an logische Variablen gebunden und der Ausdruck dann evaluiert. Ist die Auswertung ohne Fehler erfolgreich, ist das Ergebnis ein logischer Term, der direkt bei der Unifikation verwendet werden kann.

`has-logical-term` Wenn die Erzeugung des logischen Terms nicht funktionert hat (z. B. wenn Operatoren mit impliziter Typprüfung im Ausdruck auf ungebundene Variablen verwendet werden), ist dieses Feld `#f`, sonst `#t`. Der Wert dieses Feldes ist direkt der Rückgabewert der Methode `isInvertible`.

Ein invertierbarer Ausdruck kann mit Marken oder anderen Werten unifiziert werden und die in ihm enthaltenen ungebundenen Bezeichner werden durch die Unifikation gebunden (siehe u. a. [Kummer u. a., 2006a]).

`binding-construct` Dieses Feld ist `#t`, wenn im Ausdruck das Makro `is?` verwendet wurde, sonst `#f`. Diese Ausdrücke müssen probeweise ausgewertet werden, auch wenn nicht alle Variablen vor der Auswertung gebunden werden konnten.

Damit `is?` auch in benutzerdefinierten Makros verwendet werden kann, wird in der Makroexpansion eine spezielle Prozedur aufgerufen, die einen unspezifizierten Wert zurück gibt.

Die Expansion des Ausdrucks wird daraufhin überprüft, ob genau diese Prozedur darin enthalten ist, falls ja handelt es sich um die Expansion eines `is?`-Ausdrucks.

Der Prozeduraufruf fungiert in diesem Fall nur als spezielle Markierung.

Da es sich um einen Prozeduraufruf handelt, kann dieser nicht bei der Optimierung als redundant erkannt werden (dazu müsste die Semantik der Prozedur bei der Optimierung bekannt sein, was nicht der Fall ist).

12.4.2 Ungebundene Bezeichner

In der Datei `unbound-symbols.scm` wird ein SISC-spezifisches Verfahren zum Auffinden von ungebundenen Bezeichnern in beliebig komplexen Ausdrücken implementiert. Das Verfahren ist spezifisch für SISC, weil es im Standard kein Prädikat `bound?` gibt. Eine einfache, aber leider noch fehlerhafte, Variante dieses Prädikats könnte wie folgt aussehen:

```
(define (bound? sym env)
  (if (symbol? sym)
      (let ((bound1 (getprop sym env))
            (bound2 (getprop sym env 'unbound)))
        (if (and (not bound1)
                 (eq? bound2 'unbound))
            #f
            #t))
      (error "Not a symbol: ~s!~%" sym)))
```

Die verwendete Prozedur `symbol?` wird im Standard definiert, aber `getprop` ist bereits eine SISC-spezifische Erweiterung.

Leider verhält sich `getprop` für die syntaktischen Schlüsselwörter des Grundvokabulars exakt wie für ungebundene Bezeichner, deswegen müssen diese speziell abgefragt werden. Die korrekte Version des obigen Prädikats ist darum deutlich länger.

Einen kompletten Ausdruck auf ungebundene Bezeichner zu untersuchen ist etwas schwieriger. Zunächst wird der Ausdruck expan-

diert, wodurch alle Makros (wie etwa `let` oder `do`) auf das Grundvokabular der Implementation zurückgeführt werden. Eine Besonderheit dabei ist, dass `define` immer auf die syntaktische Variante zurückgeführt wird, deren erstes Argument ein Bezeichner ist. Weitere relevante Operatoren des Grundvokabulars sind außerdem der Operator `quote`, der die Auswertung seines Arguments verhindert, sowie `lambda` und `letrec`, die Variablen binden können. Damit ist bei allen diesen Ausdrücken die Liste der Bindungen leicht aus der Struktur des Ausdrucks zu extrahieren. Die durch den Operator gebundenen Variablen werden dann aus ihrer jeweiligen lexikalischen Umgebung entfernt bevor der Ausdruck rekursiv weiter analysiert wird.

Die Prozedur `unbound-symbols` erhält als erstes Argument eine *Liste* von Ausdrücken, fasst diese als Befehlssequenz auf und liefert eine Liste der in der Umgebung (zweites, optionales Argument) ungebundenen Symbole zurück.

Diese allgemeine Prozedur ruft `unbound-symbols/expanded` auf, deren Argument bereits expandiert sein muss und deren zweites Argument zwingend eine Umgebung sein muss.

Die erste Variante ist für interaktive Tests praktischer, die zweite wird in Scheme-Referenznetzen verwendet, da die Expansion ohnehin abgespeichert werden muss.

Ein konstruiertes und absichtlich etwas undurchsichtiges Beispiel für die Verwendung der ersten Variante:

```
(unbound-symbols '((define a 3)
                   (define k a)
                   (define (f x)
                     (let loop ((b a)
                                (c x)
                                (d g))
                       (if (< c 1)
                           b
                           (loop (cons 'q b)
                                 (- c 1)
                                 (+ d k 1)))))
                   (f 5)))
⟹ (g)
```

Lediglich 'g' ist ungebunden. In der expandierten Fassung des Ausdrucks wird q durch quote vor Auswertung geschützt, a, k und f werden durch *define* gebunden, x, b, c und d durch *lambda* und *loop* durch *letrec*.

12.5 Bindungssuche beim Scheme-Plugin für RENEW

Im Abschnitt 9.3 (S. 116) wurde eine Bindungssuche mittels KANREN-Code exemplarisch für eine Einbettung in Scheme durchgeführt. Dabei erledigte die Relation %reserve die Reservierung von Marken in einer Stelle.

In [Abelson u. a., 1985, Kapitel 4.4] wird eine Betrachtungsweise von Logikprogrammierung auf Basis von Datenströmen nahe gelegt. Die Werte der Datenbasis werden als Strom aufgefasst und jede Anfrage erzeugt neue Ströme durch filtern oder andere Modifikationen oder fasst Ströme zusammen, so dass als Ergebnis wieder ein Strom herauskommt.

Die Ausdrücke in Abschnitt 9.3 liefern nach dieser Betrachtungsweise einen Strom von möglichen Belegungen für Marken, die mit den Kantenausdrücken unifiziert und anschließend durch die Guards gefiltert werden, wobei am Ende der Strom der möglichen Bindungen der Transition herauskommt.

Das Logiksystem KANREN ist nach dem Paradigma der Ströme implementiert, welche zu Monaden spezialisiert werden. Es gibt für jedes Ziel eine Continuation für den Erfolgs- und eine für den Fehlerfall, die jeweils die Eingaben weiterverarbeiten bzw. die Anfrage abbrechen.

Diese Paradigmenwahl ist für ein in Scheme geschriebenes Logiksystem kaum verwunderlich, sie erklärt sich aus der von Drew McDermott und Gerald Jay Sussman geäußerten Kritik am automatischen Backtracking der Sprache PLANNER und den daraus resultierenden Versuchen, aus denen letztlich Scheme hervorgegangen ist (wiedergegeben in [Sussman und Steele, 1998])

Die RENEW-Unifikation erfüllt nach dieser Darstellungsweise für die Scheme-Referenznetze die Funktion der Relation `%reserve`, so dass der scheme-spezifische Teil der Bindungssuche im wesentlichen die folgenden Aufgaben zu erledigen hat:

1. Unifikation von Marken mit Ausdrücken an eingehenden Kanten und binden der ungebundenen Bezeichner
2. Weitergabe der Unifikationsergebnisse an den Simulator
3. Einführung von benutzerdefinierten Bindungen

Die folgenden Unterabschnitte erläutern die dazu nötigen Schritte.

Das bei der Unifikation in RENEW verfolgte Paradigma beruht aber nicht auf Strömen, sondern auf *explizitem* Backtracking, wobei zum erfolgreichen Backtracking der Zustand von Objekten in Objekten vom Typ `StateRecorder` gespeichert werden muss.

Das von RENEW vorgegebene Verhalten muss bei dem gewählten Ansatz auf der Ebene der `Expression`-Objekte zwingend eingehalten werden, wenn die Bindungssuche erfolgreich sein soll. Die Alternative mit größerer Flexibilität hätte in einem dichter am Simulator liegenden Ansatz bestanden (vgl. aber Abschnitt 12.1 (S. 140)).

12.5.1 Unifikation von Termen

Der RENEW-Simulator fordert die Unifikation zweier Werte miteinander, oder eines Ausdrucks mit einem Wert, oder zweier Ausdrücke miteinander an. Aus diesem Grund implementieren sowohl die Objekte vom Typ `<unifiable-s-expression>` als auch vom Typ `<unifiable-value>` und der abgeleiteten Klasse `<collection-value>` (jeweils in gleichnamigen Dateien) das Interface `SchemeSilentlyUnifiable`, welches wiederum das Interface `de.renew.unify.SilentlyUnifiable` erweitert (im `Simulator`-Plugin).

Unifikation erfolgt zentral über die Methode `unify` der Klasse `de.renew.unify.Unify`. Die oben genannten Interfaces wurden geschaffen, damit nicht für jede weitere Klasse ein Spezialfall in die Methode `unifySilently` eingefügt werden muss. Über die Fallunterscheidung dieser Methode in `Unify` wird die gleichnamige Methode aller Scheme-Werte und Ausdrücke aufgerufen, deren aggregiertes Proxy-Objekt direkt oder indirekt von `<scheme-silently-unifiable-proxy>` abgeleitet wurde.

In der Wrapper-Hierarchie führt dies zum Aufruf einer der vier `unify-silently` Methoden[1], die wiederum die in der Datei `unify/unify.scm` definierten Prozeduren `unify-value/value`, `unify-value/expression` oder `unify-expressions` aufrufen. Bei erfolgreicher Unifikation werden die im Unifikator enthaltenen lokalen Variablen mit der Prozedur `inject-unifier!` (`backlinks+listeners.scm`) aktualisiert. Das genaue Verfahren zur Propagation der Ergebnisse soll aber im Abschnitt 12.5.2 (S. 155) erläutert werden.

Die Unifikation selbst geschieht in den drei Unifikationsprozeduren wie folgt:

Ein Abbruch der Unifikation passiert jeweils indem eine Ausnahme vom Typ `de.renew.unify.Impossible` geworfen wird, wobei eine solche Ausnahme nur geworfen werden darf, wenn klar ist, dass die Unifikation unter Verwendung der bisher bekannten Werte nicht funktionieren kann. Sie darf insbesondere nicht geworfen werden, wenn lediglich Informationen fehlen.

1. Bei `<unifiable-s-expression>`-Objekten
   a) muss geprüft werden, ob sie überhaupt invertierbar sind. Falls nicht, ist dies ein Fehler und die Unifikation wird abgebrochen.
   b) Andernfalls werden die Felder `logical-term` und `logical-variables` ihres `<expression-analysis>`-Objekts für die Unifikation verwendet und

[1] je eine für jede mögliche Kombination zweier Werte, die entweder der Klasse `<unifiable-value>` oder `<unifiable-s-expression>` angehören

   c) aus dem Feld *subst* des Ausdrucks selbst die bisher gefundenen Substitutionen extrahiert.
   d) Im Fall von zwei <unifiable-s-expression>s werden die entsprechenden Felder einfach mit *append* zusammengefügt, da KANREN bei der Unifikation Redundanzen entfernt.

2. Dann werden beide Seiten mit der KANREN-Prozedur `unify` unter Verwendung der oben erhaltenen Werte unifiziert.

3. Dabei kommt entweder `#f` für einen Fehlschlag heraus, in welchem Fall die Unifikation abgebrochen wird,

4. oder eine „Substitution“, eine Datenstruktur in der die logischen Variablen noch direkt enthalten sind.

5. Sofern die Substitution nur noch vollständig reifizierte Variablen enthält, was mit entsprechenden KANREN-Prädikaten getestet werden kann, werden aus ihr die entsprechenden Bindungen extrahiert.

6. Andernfalls wird die erhaltene *partielle Substitution*[2] im Feld *subst* der beteiligten Ausdrücke abgespeichert. Damit gegebenenfalls noch ein explizites Backtracking durch den RENEW-Simulator durchgeführt werden kann, muss aber vorher der aktuelle Zustand der Substitution im `StateRecorder` gespeichert werden.

   Dies geschieht mit Hilfe eines Objekts vom Typ `de.renew.unify.StateRestorer`, dem über einen Java-Proxy eine beliebige Prozedur als Implementation der Methode `restore` mitgegeben wird. Außerdem werden `StateRestorer` verwendet um den Wert eines Ausdrucks sowie allgemeine Evaluationsparameter zurückzusetzen.

[2] hier als Kurzform für Substitutionen die noch ungebundene Variablen enthalten

Alle zur Unifikation benötigten Informationen wurden bereits bei der Analyse der Ausdrücke im `<expression-analysis>`-Objekt angelegt. Das Aufbewahren der *partiellen Substitionen* im Feld `subst` ist lediglich eine kleine Optimierung die nur in seltenen Fällen greift, weil in den meisten Fällen auf der Java-Seite bereits die entsprechende Information im `VariableMapper` bzw. in den `Unknown`-Objekten vorliegt. Außerdem werden Ausdrücke mit identischen Variablen über das im Abschnitt 12.5.2 erklärte Verfahren miteinander verknüpft.

Es kostet aber kaum Aufwand, die partiellen Substitutionen trotzdem zu verwenden, welche die Prozedur `unify` ohnehin ausliefert, und bringt bei komplexen `is?`-Ausdrücken oder synchronen Kanälen tatsächlich den Effekt, dass insgesamt weniger Unifikationsversuche durchgeführt werden müssen.

Hier liegt eine Verdoppelung von Funktionalität vor, die auch erwartet ist. Durch das auf der Scheme- und auf der Java-Seite doppelt ablaufende Unifikationsverfahren werden notgedrungen einige wenige Informationen doppelt generiert und berücksichtigt. Es dürfte aber eher selten vorkommen, dass Schritte auf beiden Seiten doppelt vorgenommen werden, weil die meisten Ausdrücke an Transitionen und Kanten eher einfach sind (am häufigsten treten einfache Variablen auf) und in solchen Fällen gar keine partiellen Substitutionen entstehen.

12.5.2 Backlinks und Listeners

Die bei der Übersetzung des Netzes generierten Objekte der Klasse `<unifiable-expression>` werden zunächst nur analysiert und erhalten noch keinen Wert. Wenn die Methode `startEvaluation` aufgerufen wird, wird zunächst ein Objekt des Typs `<unifiable-s-expression>` erzeugt, welches nur dann einen Wert erhält, wenn es bereits ausgewertet werden kann. Außerdem erhält dieses Objekt das im `<unifiable-expression>`-Objekt gespeicherte `<expression-analysis>`-Objekt und auch den `<scheme-compiler>` als Instanzenvariablen.

Der erhaltene Wert ist in jedem Fall ein Objekt vom Typ `<unifiable-value>` oder `<collection-value>`, letzteres ist bei flexiblen Kanten nötig (siehe Abschnitt 12.6 (S. 166)). Wenn der Ausdruck bereits beim Aufruf von `startEvaluation` ausgewertet werden kann, wird der Wert direkt zurückgegeben, ansonsten ergibt er sich aus der Unifikation.

Wenn die sofortige Auswertung scheitert, wird eine Variable (`de.renew.unify.Variable`) an den Simulator zurückgegeben, die zunächst einen `Unknown`-Wert erhält. Das RENEW-Unifikationsverfahren kommuniziert nun mit diesen Variablen über ein System von „Backlinks“ und „Listeners“. Dieses wird für Scheme-Referenznetze in der Datei `backlinks+listeners.scm` implementiert.

Backlink

Ein **Backlink** ist dabei eine Referenz auf ein anderes Objekt, welches den gleichen Wert bezeichnet. Ein Backlink ist eine Eigenschaft eines Objekts, welches das Interface `de.renew.unify.Referable` implementiert, und zeigt auf ein Objekt vom Typ `de.renew.unify.Reference`, wobei jede `Variable` (oder allgemeiner jedes Objekt vom Typ `de.renew.unify.Referer`) eine solche Referenz aufweist und Referenzen wiederum `Referer` und `Referable` beidseitig miteinander verknüpfen.

Da die Ausdrücke in Scheme-Referenznetzen bei der Auswertung Variablen zurückliefern, müssen sie `Referable` sein, also die Methoden `addBacklink` und `occursCheck` implementieren. Wenn die Ausdrücke nun durch Unifikation einen Wert erhalten, geben sie diesen an die Objekte in der Liste der Backlinks weiter. Durch diesen Mechanismus wird die Kommunikation zwischen Scheme-Werten und Variablen des RENEW-Unifikationsalgorithmus realisiert.

Listener

Ein **Listener** ist wiederum ein Objekt vom Typ `de.renew.unify.Notifiable`, das die Methode `boundNotify` implementiert. Variablen erhalten Listener, deren `boundNotify`-Methoden aufgerufen werden, sobald die Variablen gebunden sind. Über einen geeigneten Java-Proxy kann ein Listener mit einer beliebigen Scheme-Prozedur implementiert werden, in der Praxis sind aber nur zwei konkrete Fälle zu berücksichtigen.

Abbildung 12.5.2 auf der nächsten Seite zeigt ein Beispielnetz mit der dazu gehörigen Struktur aus Backlinks und Listenern:

Da es sich bei den Ausdrücken `x` und `y` an den eingehenden Kanten um nackte Variablenauswertungen handelt, werden statt neu generierter Variablen gleich geeignete lokale Variablen (Klasse `de.renew.expression.LocalVariable`) angelegt und im `VariableMapper` angemeldet (bzw. dort entnommen). Jede dieser lokalen Variablen enthält mit der Prozedur `add-unifier!` einen Listener, der den Wert der Variablen mit dem Ausdruck unifiziert, falls die Variable (z. B. durch Unifikation mit einer Marke) einen Wert erhält. Außerdem erhält jeder Ausdruck auch einen Backlink, der es wiederum erlaubt die Variable zu aktualisieren, wenn der Ausdruck einen Wert erhält. Für das Beispielnetz sind nicht beide Richtungen nötig, da `x` und `y` in jedem Fall zuerst mit Marken unifiziert werden, es gibt aber komplexere Fälle, bei denen der Informationsfluss in die andere Richtung verläuft. Diese Fälle von vornherein zu erkennen wäre sehr schwierig, weswegen immer beide Richtungen berücksichtigt werden (also sowohl Backlinks als auch Listener für Ausdrücke an Kanten vergeben werden).

Der Ausdruck `(+ x y)` wiederum ist ein komplexer Ausdruck, der zum Einen eine Variable benötigt in der sein eigener Wert verzeichnet wird. Die Variable erhält hier wiederum einen Unifier-Listener, obwohl das in diesem konkreten Fall nicht sinnvoll ist (die Implementation fängt diesen Fall aber noch nicht ab) und einen Backlink. Über den Backlink wird der Wert des Ausdrucks `(+ x y)` letztendlich an den Simulator weitergegeben. Grundsätzlich gilt: wenn alle beteiligten Objekte vom Typ `de.renew.unify.Variable` erfolgreich gebunden werden konnten, kann die Transition schalten.

Der Ausdruck `(+ x y)` enthält zum Anderen aber auch lokale Variablen, die wiederum über den `VariableMapper` mit jeweils identischen Objekten vom Typ `Variable` identifiziert werden. Um den Ausdruck an der ausgehenden Kante auswerten zu können, müssen die lokalen Variablen mit der Prozedur `add-value-updater!` je einen weiteren Listener erhal-

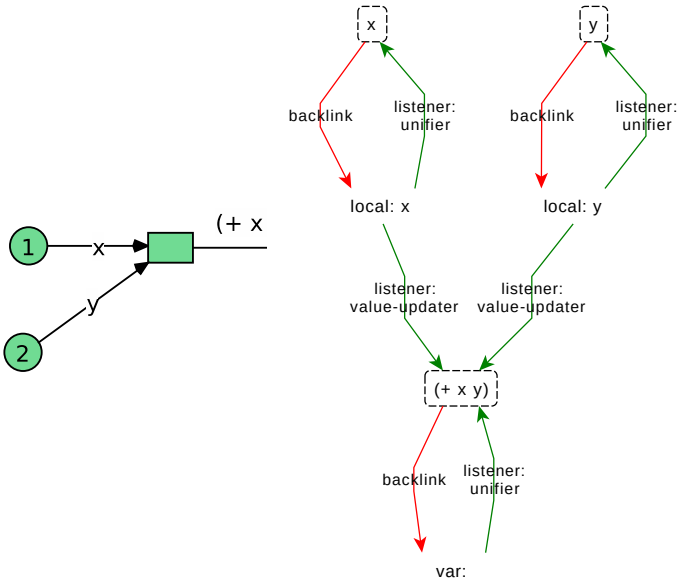

Abbildung 12.2: Beispiel: Netz mit Struktur der Backlinks und Listener

ten, der diesmal eine andere Aufgabe erfüllt: beim Aufruf der `boundNotify` Methode werden alle bisherigen Bindungen aus dem `VariableMapper` in eine sekundäre Umgebung injiziert und der Ausdruck wird ausgewertet. Ist die Auswertung erfolgreich, werden die Backlinks aktualisiert wodurch in diesem Fall der Wert `3` für den Ausdruck `(+ x y)` gebunden wird.

Der Wert eines `<unifiable-s-expression>`-Objekts wird zusätzlich im Objekt selbst gespeichert, damit er für spätere Auswertungen in der gleichen Bindungssuche bekannt ist. Damit der Wert zurückgesetzt werden kann, wird vor dem Speichern des Wertes ein `StateRestorer`-Objekt im aktuellen `StateRecorder` hinterlegt.

Guards: Die Berücksichtigung von Guards ist nun trivial: Bei der Übersetzung von Ausdrücken an Transitionen wurde ein Flag „Guard“ gesetzt. Wann immer ein solcher Ausdruck einen Wert erhält wird geprüft, ob er `#f` ist, und falls ja führt eine `Impossible`-Exception zum Abbruch der Unifikation.

Dies kann genau an drei Stellen geschehen,

1. in der Methode `startEvaluation` selbst, sofern der Ausdruck direkt ausgewertet werden kann,

2. im Listener der mit `add-value-updater!` und

3. im Listener der mit `add-unifier!` hinzugefügt wurde,

da an genau diesen drei Stellen ein Ausdruck einen Wert erhalten kann.

12.5.3 Vollständig und Gebunden

Damit der Simulator entscheiden kann, wozu die erhaltenen Werte verwendet werden können, müssen die Werte vom Typ `Unifiable` außerdem die Methoden `isComplete` und `isBound` korrekt implementieren. Dazu sei [Kummer, 2002, S.307] zitiert:

Definition 5. *Ein Objekt heißt unwandelbar, wenn es durch Unifikationen nicht mehr verändert werden kann. Eine Unbekannte ist nie unwandelbar, sie kann stets mit einem Java-Objekt unifiziert werden und erhält dadurch einen neuen Wert. Alle anderen unifizierbaren Objekte können unwandelbar werden, sobald keine Unbekannte mehr durch sie direkt oder indirekt referenziert wird. Der Zustand eines Objekts kann über die Java-Methode* `isComplete()` *abgefragt werden.*

Als gebunden gilt ein Objekt, wenn es ein vollständig ausgewertetes Java-Objekt ist. Dies gilt insbesondere nicht für Berechnungsobjekte, weil sie lediglich Platzhalter für eine noch auszuführende Operation darstellen. Damit kommen als gebundene Objekte nur noch gewöhnliche Java-Objekte und Tupel, deren sämtliche Komponenten gebunden sind, in Frage. Die Java-Methode `isBound()` *wertet den Zustand eines Objekts diesbezüglich aus.*

In [Kummer u. a., 2006a, Rel 2.1, 3.2.6] findet sich eine bedeutend knappere Definition[3]:

Definition 6. *Ein Wert ist vollständig, wenn er, oder alle durch ihn referenzierten unifizierbaren Objekte, keine Unbekannten enthält. Ein vollständiger Wert ist gebunden, wenn er keine Berechnungsobjekte enthält.*

Momentan werden in der Implementation noch keine Actions in Scheme-Referenznetzen unterstützt. Da die Semantik von Actions in Scheme-Referenznetzen sich aber auch gravierend von der in Java-Referenznetzen unterscheiden soll, werden vermutlich gar keine `Calculator`-Objekte direkt in Marken referenziert, sondern Werte vom (SISC-)Typ `<promise>`.

Daraus ergibt sich folgende Implementation der Methode `complete?` für die Klasse `<unifiable-s-expression>`:

[3] Originaltext: A value is complete if it contains no unknowns, even nested within a multitude of unifiable objects. A complete value is bound if it contains no calculators.

```
(define-method (complete? (<unifiable-s-expression> u))
  (or (:has-value u)
      (null? (:unbounds (:expression-analysis u)))))
```

Und `bound?`:

```
(define-method (bound? (<unifiable-s-expression> u))
  (:has-value u))
```

Beide Methoden geben für `<unifiable-value>` (und `<collection-value>`) lediglich `#t` zurück.

12.5.4 Einführen von Bindungen

Die Implementation sowohl der Bindungsoperatoren als auch des Konstruktors `new` und der Prozedur `this` befindet sich in der Datei `net-bindings.scm`.

Module und Parameter

Die Operatoren `bind` und `is?` (sowie später auch `new`, `this` und `action`) benötigen Zugriff auf den aktuellen Zustand des Simulators in Form der `VariableMapper`- und `StateRecorder`-Objekte (`action` wird auch den `CalculationChecker` benötigen). Diese Objekte sollen aber nicht in der primären oder sekundären Umgebung zu sehen sein, obwohl die Bindungen der Operatoren selbst sichtbar sein müssen.

Das in SISC verwendete Modulsystem auf Basis von PSYNTAX bietet die Möglichkeit die sichtbaren Bindungen auf diese Weise zu kontrollieren.

Zusätzlich wird die Fähigkeit von SISC verwendet, auch Definitionen mit *`eval`* in einer Umgebung auszuwerten. Auch die aktuelle Variablenbelegungen, wie z. B. Prozeduren oder Parameter-Objekte dürfen mittels `unquote` im auszuwertenden Ausdruck verwendet werden. Dadurch können außerhalb der Umgebung definierte Prozeduren in den definierten Operatoren verwendet werden, ohne dass sie selbst in der Umgebung sichtbar werden.

In der primären Umgebung wird also die Definition eines Moduls evaluiert, das die benötigten Operatoren definiert und exportiert. In

diesem Modul wird aber auch eine Prozedur definiert, die Zugriff auf den Simulator in Form der oben genannten Objekte bietet. Diese Prozedur wird nicht exportiert aber von den exportierten Operatoren verwendet.

Eine frühere Version des Plugins musste nach dem Erzeugen der sekundären Umgebung nochmals `eval` aufrufen, um die aktuellen Werte für `VariableMapper`, `StateRecorder` und `CalculationChecker` in ein Modul zu injizieren, damit diese von den Operatoren in der sekundären Umgebung verwendet werden können.

In der aktuellen Version wird dieses Problem mit Hilfe von dynamischen Parametern [SRFI-39, 2003] gelöst. Dynamische Parameter sind Objekte mit dynamischem Gültigkeitsbereich, im Gegensatz zu der sonst in Scheme üblichen lexikalischen Sichtbarkeit von Bindungen. Durch einzelne Objekte mit dynamischem Gültigkeitsbereich kann z. B. das Verhalten einer Prozedur vor ihrem Aufruf durch bestimmte Konfigurationsparameter kontrolliert werden.

Die Prozedur

```
get-current-mapper-recorder-checker+compiler
```

gibt die Werte von vier Parametern zurück, die vor dem ersten Aufruf der Methode `startEvaluation` den Wert `#f` erhalten, aber beim Aufruf der Methode `startEvaluation` temporär auf die Werte gesetzt werden, die der Methode übergeben wurden. Dadurch kann der gesamte von der Prozedur aufgerufene Code die aktuellen Werte verwenden, was auch für die Operatoren in den Anschriften selbst gilt.

Die sekundäre Umgebung muss darum nach dem Aufruf der Übergangsprozedur nicht mehr angepasst werden.

`bind`

Der Operator `bind` ist als `syntax-case`-Makro implementiert. Das einzige Pattern `(_ name value)` ist mit dem syntaktischen Guard

`(identifier? #'name)` versehen, der sicherstellt, dass nur ein Bezeichner auf der linken Seite verwendet werden kann[4].

1. Das generierte Programm prüft dann zunächst, ob der Bezeichner bereits anderweitig gebunden wurde und wirft eine `RunTimeException` falls das der Fall ist. Der `bind`-Operator muss sich also die Ausdrücke merken, die auf der rechten Seite verwendet werden. Die rechten Seiten werden darum in anonymen Variablen im `VariableMapper` abgelegt. Die Namen dieser anonymen Variablen dürfen keine Kollisionen mit legalen Bezeichnern erzeugen und bestehen darum ausschließlich aus arabischen Ziffern.

2. Als nächstes wird für die linke Seite ein `<unifiable-expression>`-Objekt angelegt und ausgewertet,

3. Anschließend wird versucht, ob sich in der aktuellen Umgebung die rechte Seite direkt auswerten lässt.

4. Falls das funktioniert, wird der Ausdruck direkt an den Wert der rechten Seite gebunden.

5. Für den Fall, dass die rechte Seite ebenfalls noch nicht gebundene Variablen enthält, wird eine Ausnahme vom Typ `Impossible` geworfen und die Unifikation damit abgebrochen. Das ist in diesem Fall richtig, weil lediglich die linke Seite eines `bind`-Ausdrucks ungebunden sein darf, wenn der Ausdruck ausgewertet wird.

6. Ein Sonderfall ergibt sich, wenn die rechte Seite eines `bind`-Ausdrucks eine Prozedur ist, da Prozeduren ebenfalls ungebundene Variablen referenzieren können. Für diesen oder ähnliche Fälle muss für die rechte Seite ebenfalls

[4] Im Folgenden ist bei einem zweistelligen Bindungs- oder Vergleichsoperator mit „linker Seite“ das erste und mit „rechter Seite“ das zweite Argument des Operators gemeint.

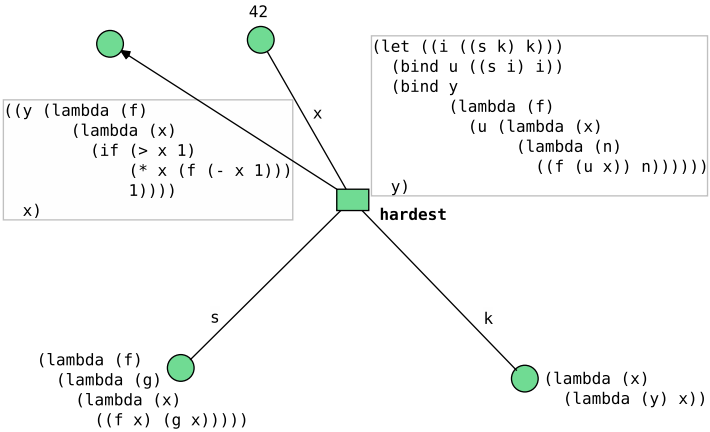

Abbildung 12.3: Komplexes Beispiel für einen `bind`-Ausdruck

ein `<unifiable-expression>`-Objekt angelegt, ausgewertet und mittels `add-unifier!` mit der linken Seite verknüpft werden, damit die Prozedur angepasst werden kann, sobald in ihr referenzierte Bezeichner gebunden werden (siehe Abschnitt 12.5.2 (S. 155)).

Da die Expansion des Makros ein *`define`*-Ausdruck ist, werden die in einem lokalen Gültigkeitsbereich enthaltenen `bind`-Ausdrücke zu *`letrec`*-Klauseln expandiert (siehe [R^{5}RS, 1998]).

Nachdem der `bind`-Ausdruck erfolgreich ausgewertet wurde, ist die Variable sowohl im RENEW-Simulator als auch in der jeweiligen sekundären Umgebung gebunden, so dass nachfolgende Ausdrücke im gleichen Gültigkeitsbereich ihren Wert verwenden können.

Das Beispiel in Abbildung 12.3 verwendet zwei `bind`-Ausdrücke in einem lokalen Gültigkeitsbereich. Die Variable `u` wird in der rechten Seite der Definition von `y` verwendet. Außerdem wird `y` durch einen *`lambda`*-Ausdruck bestimmt, der u. U. aktualisiert werden muss, nachdem er bereits festgelegt wurde. Damit treten in dieser einen Beschriftung alle gerade vorgestellten Fälle ein.

`is?`

Da auf beiden Seiten eines `is?`-Ausdrucks beliebige Ausdrücke erlaubt sind, kann der Operator mit einem *`syntax-rules`*-Transformer nach R^{5}RS implementiert werden. Gleich zu Beginn der Expansion wird der Prozeduraufruf zur Markierung des `is?`-Ausdrucks eingefügt, da die Makroexpansion linear nach dem Aufruf durchsucht werden muss und dieser möglichst schnell gefunden werden sollte. Da im Regelfall jedoch der gesamte Ausdruck durchsucht werden muss, weil die meisten Ausdrücke kein `is?` enthalten, ist es günstig, dass dieser Schritt bereits beim Übersetzen des Netzes einmalig vorgenommen wird.

1. Für jede Seite wird nun durch das generierte Programm ein `<unifiable-expression>`-Objekt erzeugt und probeweise ausgewertet.

2. Dann wird zunächst geprüft ob beide Seiten bereits in der sekundären Umgebung gebunden sind.

3. Wenn ja, gibt der Ausdruck `#t` zurück, falls sie identisch (`equal?`) sind, sonst `#f`.

4. Falls eine oder beide Seiten aber nicht direkt gebunden werden können, was der Regelfall sein sollte, werden die dazu gehörigen `<unifiable-s-expression>`-Objekte mittels `Unify.unify()` unifiziert.

5. Falls das auch nicht funktioniert (weil erst noch ungebundene Variablen außerhalb des `is?`-Ausdrucks selbst gebunden werden müssen) werden die beiden Seiten mittels `add-value-updater!` miteinander verknüpft (siehe Abschnitt 12.5.2 (S. 155)).

6. Da Ausdrücke, die Expansionen von `is?` enthalten, anders als alle anderen Ausdrücke *probeweise* auch mit ungebundenen Variablen ausgewertet werden, wird nach ihrer Auswertung geprüft, ob alle Variablen gebunden werden konnten.

7. Falls das nicht der Fall ist, wird anschließend der vorher gespeicherte Zustand aus dem `StateRecorder` wiederhergestellt.

12.6 Flexible Kanten

Ausdrücke an flexiblen Kanten dürfen im RENEW-Simulator bisher Arrays, Listen, d.h. in diesem Fall Instanzen der Klasse `de.renew.unify.List`, oder Werte des Typs Collection (`java.util.Collection`) liefern.

Die Ausdrücke an flexiblen Kanten in Scheme-Referenznetzen sollen Scheme-Listen als Werte liefern. Trotz des Namens lässt sich die Klasse `de.renew.unify.List` nur schwer an die Verwendung mit Scheme-Listen anpassen, da bei jeder Verwendung die Liste explizit konvertiert werden müsste.

Das Interface `Collection` fordert dagegen lediglich die Implementierung weniger Methoden, die sich sehr leicht an Scheme-Listen anpassen lassen, darum wird die Klasse `<unifiable-value>` zu `<collection-value>` erweitert.

Die Implementation der `Collection`-Methoden mit einer Scheme-Liste ist so einfach, dass die meisten Implementationen nicht länger als eine Zeile sind.

Der RENEW-Simulator verwendet ausschließlich die Methode `iterator`, die eine Implementation des Interfaces `java.util.Iterator` liefert. Dieser wiederum ist mit einem Java-Proxy implementiert der den Zustand des Iterators in seinem lokalen lexikalischen Gültigkeitsbereich hält.

Dieses Verfahren zur Implementation von Interfaces in SISC ist in diesem Fall eine leichtgewichtige Alternative zu der komplexen Klassenhierarchie aus Abschnitt 12.3 (S. 143).

12.7 Netzexemplare

Netzexemplare können an allen Netzelementen, aber nicht im Deklarationsknoten erzeugt werden. Außerdem darf nicht zu früh versucht werden, ein Netzexemplar zu erzeugen. Da Ausdrücke bereits beim Erzeugen der logischen Terme ausgewertet werden, müssen die Prozeduren `this` und `new` prüfen, ob ein gültiges `VariableMapper`-Objekt vorhanden ist, bevor sie einen Wert liefern.

Beide Operatoren werden als Prozeduren implementiert, da sie keine in ihren Parametern auftretenden Variablen binden.

12.7.1 `this`

Die versteckte lokale Variable „*this*“ ist in jedem Netzexemplar in RENEW gebunden, also reicht ein Zugriff auf den `VariableMapper` aus um das aktuelle Netzexemplar zu erhalten.

12.7.2 `new`

Die Erzeugung eines Exemplars eines gegebenen Netzmusters geschieht wie folgt:

```
(define (new-net net-name)
  (build-instance
   (for-name (java-null <de.renew.net.net>)
             (->jstring net-name))))
```

Wobei `for-name` und `build-instance` generische Prozeduren für die Java-Methoden `de.renew.net.Net#forName()` respektive `de.renew.net.Net#buildInstance()` sind. Wie bei `this` auch muss der in der Netzanschrift verwendete Operator `new` vorher aber noch prüfen, ob ein gültiger `VariableMapper` zur Verfügung steht.

Die Verwendung von `buildInstance` ist an dieser Stelle nicht ganz unproblematisch, da diese Methode eigentlich für Methodenaufrufe auf Java-Netzinstanzen außerhalb des Simulators gedacht ist.

Trotz der Warnung im Quellcode der Methode verhalten sich aber auch die in Stellen oder an Kanten erzeugten Netzexemplare wie erwartet.

Unter Umständen wird versucht unendlich viele Netzexemplare des gleichen Netzmusters zu erzeugen. Das solle aber nur dann der Fall sein, wenn ein Netzexemplar in seiner initialen Markierung eine Referenz auf ein Exemplar des gleichen Netzmusters erzeugt.

Die Erzeugung von Netzexemplaren aus Scheme-Netzbeschriftungen heraus gelingt überraschend einfach, da die meiste Arbeit bereits vom RENEW-Simulator erledigt wird. An dieser Stelle, wie auch bei den synchronen Kanälen, zahlt sich die Entscheidung aus, als erste praktische Umsetzung von Scheme-Referenznetzen ein RENEW-Plugin zu implementieren.

12.7.3 Behandlung der Werte

Eine zusätzliche Anforderung ergibt sich noch aus der Behandlung von Netzexemplaren durch die grafische Simulation und an synchronen Kanälen.

Wenn eine Stelle im grafischen Simulator ein Netzexemplar enthält, wird zunächst nur der Name des Netzmusters und sein `ID`-Wert (in der Regel eine Nummer) angezeigt. Dieses Element kann aber angeklickt werden und in einem neuen Fenster wird dann die grafische Simulation des Netzexemplars mit seiner aktuellen Markierung angezeigt.

Netzexemplare in Marken erhalten im grafischen Simulator automatisch ein solches `ClickHandle`.

Wenn Netzmuster in Scheme erzeugt werden, werden sie aber zunächst in Scheme-Objekte verpackt, damit sie durch Scheme-Operatoren bearbeitet werden können. Der Simulator kann dann nicht mehr erkennen, dass es sich um Netzexemplare handelt. Um dieses Manko zu beheben eröffnen sich zwei Möglichkeiten: Entweder kann der Java-Code des Simulators so angepasst werden, dass er Netzexemplare in Scheme-Objekten erkennt, oder überall dort wo Netzexemplare in Scheme-Objekten dem Simulator übergeben

werden, werden sie direkt als Java-Objekte übergeben. Die zweite Möglichkeit erscheint deutlich einfacher umzusetzen:

Ähnlich wie bei der Behandlung von Guards muss dazu eine Abfrage überall dort erfolgen, wo eine RENEW-Variable ihren Wert aus einem Scheme-Ausdruck erhält, also in der Methode `start-evaluation`, sowie in den Prozeduren `unify-backlinks` und `inject-unifier!` (Datei `backlinks+listeners.scm`) und in der `next()` Methode des Iterators eines `<collection-value>`-Objekts.

Damit werden einzelne Netzinstanzen in diesem Sinne aktiv und können auch an synchronen Kanälen verwendet werden.

Damit lässt sich aber nicht realisieren, dass Netzinstanzen auch *innerhalb* von Listen oder anderen Scheme-Datentypen aktiv sind (was bei den Listen und Tupeln in Java-Referenznetzen der Fall ist).

Bevor ein Netzexemplar an einem synchronen Kanal verwendet werden kann, muss es allerdings ohnehin aus der Datenstruktur, in der es eventuell vorher enthalten war, entnommen werden. Außerdem können alle im Simulator vorhandenen Netzexemplare auch im „Remote Server“ Dialog im Menü „Simulation“ angezeigt werden. Da der `ID`-Wert eines Netzes dort und auch in der Darstellung der Scheme-Liste angezeigt wird, lässt sich das betreffende Exemplar also über diesen Dialog in jedem Fall auffinden. Darüber hinaus können die in einer Scheme-Liste enthaltenen Netzexemplare leicht mit einer ausgehenden flexiblen Kante „ausgepackt“ werden.

Die Implementation dieser zusätzlichen Funktionalität erhält damit geringe Priorität.

Entgegen der in Kapitel 8 (S. 95) gegebenen Reihenfolge erfolgte die Implementation von Netzexemplaren direkt nach der Fertigstellung der Bindungsoperatoren, hauptsächlich weil sie so einfach war.

12.8 Spezielle Operatoren

Demgegenüber musste für die speziellen Operatoren der Parser für Netzanschriften erweitert und andere Veränderungen am Compiler

vorgenommen werden. Die bereits im vorigen Abschnitt diskutierte Sonderbehandlung für Netzexemplare ist wiederum für die korrekte Behandlung von Netzexemplaren an synchronen Kanälen wichtig.

12.8.1 Parsing

Bisher konnten alle Netzbeschriftungen ausschließlich mit der Prozedur *read* eingelesen und mit `sc-expand` geprüft werden. Dadurch war es möglich reine Scheme-Coloured-Nets in RENEW einzubinden, ohne dass dafür ein eigener Parser geschrieben werden musste.

Wie bereits erklärt haben die an Transitionsbeschriftungen erlaubten speziellen Auszeichnungen eine S-Expression-ähnliche Syntax, aber unter Verwendung der Tokens '`{`' und '`}`' zur Markierung des jeweiligen Blocks. Da diese Tokens zum Einen im Standard reserviert sind, zum Anderen aber in SISC als Teil eines Bezeichners erlaubt sind, können sie leider nicht mehr mit *read* eingelesen werden.

Der sehr einfache Parser in der Datei `read-transition-inscriptions.scm` verwendet das SISC-Modul `string-io`, eine Erweiterung von [SRFI-6, 1999] um die Teilausdrücke mittels *read* einzulesen und außerdem [SRFI-13, 2000] für einige String-Operationen. Da die verwendeten Prozeduren beliebige Werte und zusätzlich einen oder mehrere Statuswerte zurückgeben, wird *values* zur Rückgabe mehrerer Werte in Verbindung mit [SRFI-8, 1999] verwendet.

Die Verwendung von *read* für die Teilausdrücke, die ja wiederum legale Scheme-Ausdrücke sein sollen, erspart zwar die Implementationsarbeit für einen eigenen spezialisierten Scheme-Parser, führt aber dazu, dass am Ende einer `<special-operation>`-Beschriftung der Fall eintreten kann, dass der Parser in bestimmten Fällen mit dem Rest des Textes neu gestartet werden muss.

Dieser Fall sollte nicht allzu häufig auftreten, obwohl mehrere Auszeichnungen und Ausdrücke in der gleichen Beschriftung vorkommen dürfen:

```
{? exchange be bf}
{! isFull bf}
{! exch (this) be bf}
```

In dieser Beschriftung wird z. B. in allen drei Fällen am Ende des Ausdrucks das Symbol `bf}` gelesen. Das Symbol am Ende einer speziellen Operation wird darum immer darauf hin untersucht, ob es mit einer schließenden geschweiften Klammer endet. In diesem Fall wird es in einen String umgewandelt, die Klammer hinten abgeschnitten und das Symbol ohne Klammer an den bisherigen Ausdruck angehängt.

Wenn lediglich solche Ausdrücke auftreten, müsste der Parser aber nicht neu gestartet werden, da der Rest des Textes direkt weiter eingelesen werden kann. Leider kann man aber nicht ausschließen, dass ein Anwender etwa das Folgende eingibt:

```
{? exchange be bf}{! isFull bf}
```

Das entspricht zwar nicht dem allgemein empfohlenen Stil für S-Expressions, da zwischen einer schließenden und einer öffnenden Klammer ein Zwischenraum stehen sollte, aber der Parser darf auf so eine Syntax nicht empfindlich reagieren.

Leider wird in der Mitte des Ausdrucks das Symbol `bf}{!` (sic!) gelesen. Nach dem Abschneiden der schließenden geschweiften Klammer wird die erste Auszeichnung zu `{? exchange be bf}` vervollständigt, der Text „`{!`“ wurde aber bereits eingelesen und gehört schon zum nächsten Ausdruck.

In diesem Fall müssen die Zeichen in den Text „zurück gelegt“ und der Parser ab der öffnenden geschweiften Klammer neu gestartet werden.

Das Ergebnis des Parsers ist eine Liste von Paaren aus Scheme-Ausdrücken und je einem booleschen Wert: `#t` für „Special Operation“ oder `#f` für „Scheme Expression“.

12.8.2 Übersetzung

Beim Erzeugen von Transitionen (Datei `scheme-formalism-transitions.scm`) stehen die speziellen

Operationen dann wieder als S-Expressions zur Verfügung.
Die Prozedur `check-transition-inscriptions` prüft Scheme-Ausdrücke wie alle anderen Beschriftungen durch Makro-Expansion.

Für spezielle Operationen wird zusätzlich die Anzahl der Parameter überprüft, wobei `manual` keine, `?` mindestens einen und `!` mindestens zwei Parameter haben muss. Der Name des Kanals wird darauf hin geprüft, ob er ein Symbol ist und die folgenden Parameter werden probeweise expandiert.

Bei der Übersetzung wird dann je nach Beschriftungstyp ein passendes Objekt zur Transition hinzugefügt. Bei `manual` handelt es sich um das Singleton-Objekt der Klasse `ManualInscription` (Paket `de.renew.net`), bei `?` und `!` jeweils um frische Objekte der Klassen `UplinkInscription` (Paket `de.renew.net`) bzw. `DownlinkInscription` (Paket `de.renew.net.inscription`), denen die Kanalnamen als Java-Strings und die Parameterlisten als `TupleExpression` (Paket `de.renew.expression`) mitgegeben werden. Die `DownlinkInscription` erhält zusätzlich das Netzexemplar als `<unifiable-expression>` und die aktuelle Transition. Das übergebene Flag `isOptional` wird (wie bei allen anderen Formalismen) auf *`false`* gesetzt.

Weitere Maßnahmen sind nicht nötig, da bereits dafür gesorgt wurde, dass Netzexemplare in einer für den Simulator akzeptablen Form erzeugt werden.

Die Unifikation der Kanalparameter erfolgt mit den bereits geschriebenen Programmteilen für Scheme-Coloured-Nets. Es ergeben sich lediglich kleinere Besonderheiten, z. B. dadurch dass eine Transition mehrere Downlinks haben kann, in denen unterschiedliche Werte an die gleichen Kanalparameter gebunden werden, oder dass ein Uplink mehrfach mit unterschiedlichen Werten für den gleichen Kanalparameter aufgerufen werden kann.

Bei allen anderen Beschriftungen stehen gleiche Symbole auch für gleiche Werte.

In Abbildung 12.4 rechts gegenüberliegend sind beide Möglich-

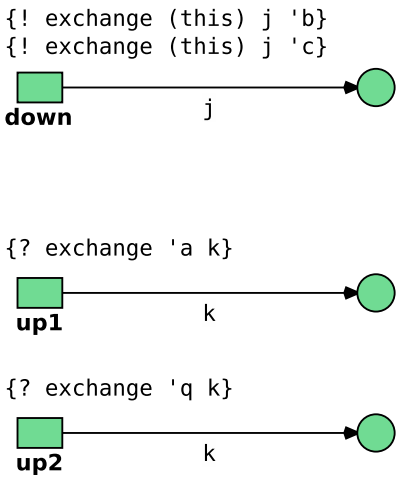

Abbildung 12.4: Gleichnamige Parameter mit unterschiedlichen Bindungen an synchronen Kanälen

keiten zu sehen. Es gibt zwei Downlinks an einer Transition und zwei Transitionen mit jeweils einem passenden Uplink. Die Auswahl des Uplinks erfolgt zufällig durch den RENEW-Simulator. Bei einem einzigen Schaltvorgang muss genau einer der beiden Uplinks je zwei mal aktiviert werden, aber es müssen immer beide Downlinks aktiviert werden. Der Bezeichner `j` kann damit *entweder* an den Wert `a` *oder* `q` gebunden werden, während `k` *sowohl* an `b` *als auch* an `c` gebunden wird.

Teil IV

Ergebnis

13 Verwendung des Scheme-Formalismus

Dieses Kapitel könnte zusammen mit den praktischen Teilen der Kapitel 5 bis 8 in englischer Übersetzung als Grundstock einer Anleitung für das Scheme-Plugin dienen.

Das Scheme-Plugin ist zur Zeit nur mit einer aktuellen Entwicklerversion 2.2 von RENEW lauffähig, da einige zentrale Klassen angepasst werden mussten. Es ist *nicht* kompatibel mit RENEW Version 2.1 oder kleiner. So bald die Version 2.2 von RENEW veröffentlicht wurde, dürfte es aber möglich sein, das Scheme-Plugin als JAR-Datei einfach in das Verzeichnis `plugins` der RENEW-Distribution zu legen. Die Installation ist damit identisch zur Installation aller anderen Plugins. Es müssen keine weiteren Klassen oder Programme installiert werden, da bereits alles nötige im JAR-Archiv des Scheme-Plugins enthalten ist.

Nach dem anschließenden Neustart des Programms `renew` befindet sich im Untermenü „Simulation → Formalisms“ der Eintrag „Scheme Net Compiler“ der ausgewählt werden muss, um Scheme-Referenznetze zu verwenden. Anschließend wird die Syntax von Beschriftungen direkt nach ihrer Erstellung überprüft und Scheme-Referenznetze können simuliert werden.

Das Scheme-Plugin ist aber noch nicht sehr benutzerfreundlich:

Sämtliche Laufzeitfehler in Scheme-Beschriftungen, die nach dem Übersetzen des Netzes auftreten, werden als „Exceptions“ auf die Konsole ausgegeben. Es ist aber ohnehin empfehlenswert, das Programm `renew` zur Verwendung mit Scheme-Referenznetzen z. B. aus dem `emacs` mit `M-x run-scheme` zu starten (benötigt die `cmuscheme`-Bibliothek, die in aktuellen Emacs-Versionen

bereits enthalten ist).

Es gibt noch keinen Befehl zur Korrektur der Einrückung und der RENEW-Editor ist nicht in der Lage passende Klammerpaare zu erkennen, also muss für komplexere Anschriften noch ein geeigneter externer Editor verwendet werden.

Es ist empfehlenswert für Beschriftungen stets den `MonoSpace`-Font und linksbündige Ausrichtung (Menüpunkt „Attributes → Text Alignment → Left“) zu verwenden. Später einmal soll dies bei der Auswahl des Scheme-Plugins automatisch geschehen.

Scheme-Referenznetze können exakt wie Java-Referenznetze im grafischen Simulator verwendet werden (siehe [Kummer u. a., 2006b]). Der Export von Shadow-Netzen wird noch nicht unterstützt.

Das Kommando `scheme` an der `renew`-Kommandozeile führt Scheme-Ausdrücke in seiner Parameterliste direkt aus, wobei allerdings mehrere Leerzeichen stets zu einem einzigen zusammengefasst werden. Wird es ohne Parameter aufgerufen, wird der RENEW-Prompt durch eine Scheme-REPL ersetzt.

In der REPL stehen alle im SISC-Handbuch [Miller und Radestock, 2006] bzw. im Standard [R^{5}RS, 1998] beschriebenen Bindungen zur Verfügung. Zusätzlich kann mit dem dynamischen Parameter `activate-debugging` dafür gesorgt werden, dass das Scheme-Plugin diverse Statusinformationen ausgibt.

Dabei ist z. Z. `3` der kleinste und `7` der höchste Level:

```
(activate-debugging 6)
```

sorgt dafür, dass je nach Komplexität der Ausdrücke im Netz etwa gut tausend Zeilen pro Schaltvorgang ausgegeben werden.

Die Scheme-REPL kann mit `(exit)` oder der Tastenkombination `Ctrl+D` (auf Unix-Systemen) verlassen werden, so dass wieder Kommandos am RENEW-Prompt verwendet werden können.

Es ist geplant auch einige der Beispiele aus dieser Arbeit mit dem Scheme-Plugin auszuliefern, damit dem Anwender zum leichteren Einstieg schon fertige Netze zum Ausprobieren zur Verfügung stehen.

14 Bewertung

Sowohl die informelle Spezifikation als auch die erste Implementation als RENEW-Plugin sind so weit gelungen, dass der Titel der Arbeit „Referenznetze mit Anschriften in Scheme“ für beide Teile gerechtfertigt ist.

In der zur Verfügung stehenden Zeit konnte aber sowohl die Implementation als auch die Spezifikation noch nicht den Reifegrad der Java-Referenznetze erreichen. Auch konnten viele interessante Erweiterungen des Konzeptes noch nicht im Rahmen dieser Arbeit umgesetzt werden.

14.1 Erreichte Ziele

Scheme-Referenznetze bieten einen Formalismus für Referenznetze mit Beschriftungen in Scheme, bei dem die Lokalität einer Transition klar aus der Syntax ihrer Beschriftungen erkennbar ist, eine klare Differenzierung zwischen Guards, Bindungen und Aktionen vorhanden ist und die einen Mechanismus bieten, mit dem der in einem Netz verfügbare Sprachumfang kontrolliert werden kann. Über den letztgenannten Mechanismus lassen sich auch schon einige der Eingangs diskutierten Nebeneffekte ausschließen (aber leider nicht alle, vgl. Abschnitt 14.2.1 (S. 188)).

Netzexemplare funktionieren in der Implementation ebenfalls bereits wie bei Java-Referenznetzen, außerdem wurden bereits einige zusätzliche Kantentypen implementiert.

Eine Aufzählung der Beispiele aus diesem Text, die noch nicht funktionieren, ist daher wesentlich kürzer als für den umgekehrten Fall:

1. Abbildung 7.5 (S. 92) funktioniert nicht weil Clear- und Inhibitorkanten noch nicht implementiert sind.
2. Abbildung 8.8 (S. 103) funktioniert nicht, weil Actions nicht implementiert sind.
3. Abbildung 8.16 (S. 112) funktioniert nicht, weil `require-library` und `require-extension` nicht verwendbar sind.
4. Abbildung 14.4 (S. 193) funktioniert noch nicht (siehe dort).

14.1.1 Scheme-Referenznetze

Die Möglichkeiten von Scheme-Referenznetzen sollen wiederum mit einem Beispiel illustriert werden, das allerdings diesmal etwas umfangreicher sein soll. Auf der Objekt-Petrinetz Webseite von Charles Lakos[1] findet sich die folgende Beschreibung der „russischen Philosophen“[2]:

> Das Problem der russischen Philosophen ähnelt dem Philosophenproblem, aber jeder russische Philosoph denkt über das Philosophenproblem nach. Wenn ein vorgestelltes Philosophenproblem in eine Verklemmung gerät, hört der entsprechende Philosoph auf zu denken und versucht mit dem Essen zu beginnen. Wenn der Philosoph aufhört zu essen, fängt er wieder an, über ein neues Philosophenproblem nachzudenken. Selbstverständlich kann dieses System beliebig tief verschachtelt werden.

[1] http://www.cs.adelaide.edu.au/~charles/OPN.html

[2] Originaltext: The Russian philosophers are like the dining philosophers, except that each Russian philosopher is thinking about the dining philosophers problem. Only when such an imagined dining philosophers problem deadlocks, will the corresponding Russian philosopher stop thinking and try to start eating. When such a Russian philosopher stops eating, he or she starts thinking about a fresh dining philosophers problem. Naturally, this system can be nested arbitrarily deeply.

Dieses Problem eignet sich gut für eine Implementation als Scheme-Referenznetz, die im Folgenden vorgestellt wird. Die Implementation verwendet ausschließlich Funktionalität, die bereits im Scheme-Plugin für RENEW funktioniert. Auch werden fast alle verfügbaren Konstruktionen verwendet, außer Reservekanten und dem Operator `is?`. Im erstellten Netzsystem werden sowohl synchrone Kanäle als auch Scheme-Konstrukte ausgiebig verwendet, aus diesem Grund erschien es trotz seiner relativen Komplexität als Beispiel am geeignetsten.

Abbildung 14.1 auf der nächsten Seite zeigt das Netzmuster `matryoshka-problem`.

Im Deklarationsknoten werden zunächst mit Hilfe des Makros `define-states` vier einfache Zustandsprozeduren erzeugt, die aus einem Philosophen und seiner Nummer eine Liste aus dem Philosophen mit seinem Kontext erstellen. Der Kontext besteht aus der Nummer und dem Zustand des Philosophen. Auch die weiteren Definitionen im Deklarationsknoten sollen eine kurze und verständliche Notation der Netzbeschriftungen ermöglichen.

Der Uplink `init` in der gleichnamigen Transition wird von einem übergeordneten Netz aufgerufen. Die so erhaltenen Parameter `num` (Anzahl der Philosophen im aktuellen Problem) und `depth` (Rekursionstiefe) werden zur lokalen Konfiguration in der Stelle **`configuration`** verwendet, außerdem wird die benötigte Anzahl nummerierter Essstäbchen in die Stelle **`chopsticks`** gelegt und eine Liste von nummerierten Philosophen im Zustand `arriving` erstellt.

Da jeder Philosoph ein eigenes Netzexemplar ist, aber Netzexemplare innerhalb von anderen Scheme-Werten im grafischen Simulator (noch) nicht anklickbar sind, werden die Netzexemplare ohne ihren Kontext zusätzlich in der Stelle ***`display`*** **`philosophers`** abgelegt. Außerdem wird die gesamte Liste aller Philosophen später benötigt um eine Verklemmung zu erkennen und dafür in der Stelle **`phils`** gespeichert.

Die Philosophen gelangen mitsamt ihrem jeweiligen Kontext in das Vorzimmer **`hall`**, wo sie darauf warten, in das Esszimmer (Stel-

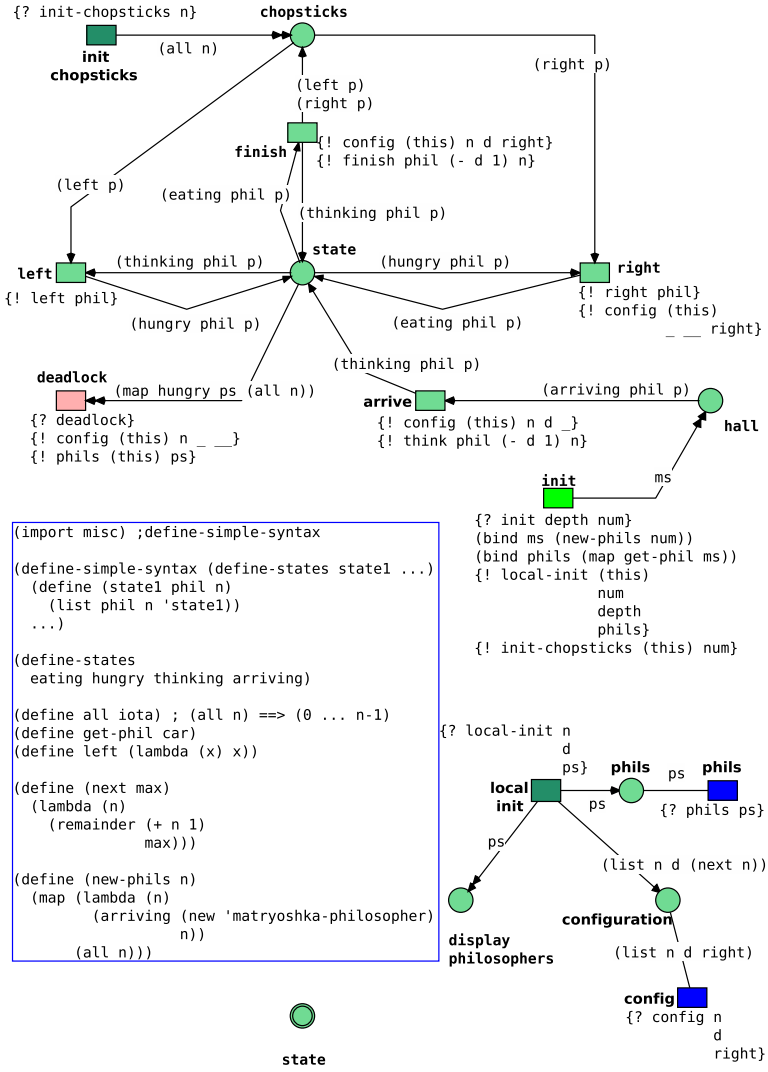

Abbildung 14.1: Russische Philosophen: Netz `matryoshka-problem`

le **state**) gebeten zu werden.

Da es nicht möglich ist, auf eine unbekannte Anzahl von Netzexemplaren gleichzeitig einen Downlink aufzurufen, muss jeder Philosoph einzeln in das Esszimmer gehen und erhält auf dem Weg zum Platz seine Anregung zum Nachdenken über den Link `think`. Dieser wird mit der Anzahl der Philosophen und der gewünschten Rekursionstiefe aufgerufen, die neben anderen Parametern über den Link `config` abgefragt wird.

Die Transitionen `left`, `right` und `finish` entnehmen nun jeweils einen Philosophen `phil` mit der Nummer `p` im entsprechenden Zustand aus der Stelle **state** und entnehmen linke oder rechte Essstäbchen mit `(left p)` oder `(right p)` aus der Stelle **chopsticks** bzw. legen sie wieder zurück. An den Transitionen `right` und `finish` muss dafür die Konfiguration über den Link `config` abgefragt werden. Die Closure `right` liefert dabei das nächste nummerierte Esstäbchen und muss dazu die Anzahl der Philosophen im Netz in ihrer lokalen Umgebung speichern, deswegen wurde diese Prozedur ebenfalls in die Stelle **configuration** gelegt.

Wenn ein Philosoph mit dem Essen fertig ist, beginnt er wieder über ein neues Philosophenproblem nachzudenken, dazu muss wieder der Link `config` aufgerufen werden. (Nicht benötigte Konfigurationsvariablen werden durch die Bezeichner _ bzw. __ angedeutet.)

Sofern alle Philosophen in der Stelle **state** im Zustand `hungry` sind, also nur ein linkes Esstäbchen in der Hand halten, ist die Stelle **deadlock** aktiviert und der Uplink kann aufgerufen werden.

Abbildung 14.2 auf der folgenden Seite zeigt einen einzelnen Philosophen (also ein Netzmuster vom Typ `matryoshka-philosopher`). Der Uplink `think` ist zwei mal vorhanden. Wenn die übergebene Rekursionstiefe gleich `0` ist, denkt der Philosoph nicht nach, sondern schläft einfach ein. Andernfalls wird ein neues Netz `m` vom Typ `matryoshka-problem` erzeugt und initialisiert. Die Transition **left-conclusion** kann nun nur schalten, wenn das Netz `m` garantiert keine lebendige

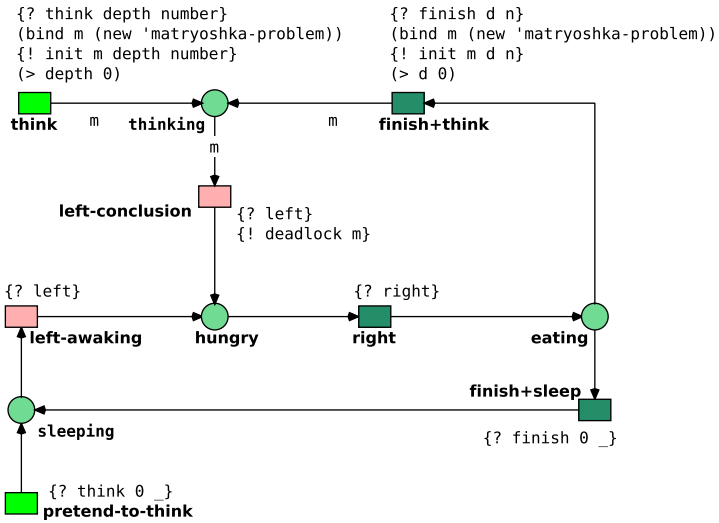

Abbildung 14.2: Russische Philosophen: Netz `matryoshka-philosopher`

Nachfolgemarkierung mehr hat, was in diesem konkreten Fall daran festgestellt wird, dass alle Philosophen darin hungrig sind.

Damit die Simulation überhaupt gestartet werden kann, braucht es noch ein Netz ohne Uplinks, welches die globalen Konfigurationsparameter für das erste Philosophenproblem setzt. Dieses ist in Abbildung 14.3 rechts gegenüberliegend zu sehen. Der Deklarationsknoten enthält außerdem eine gut gemeinte Warnung: Eine Simulation mit `5` Philosophen und Rekursionstiefe `2` benötigt immerhin mindestens 37 Netzinstanzen gleichzeitig, wenn alle Philosophen gerade denken. Die Anzahl der mindestens benötigten Netze beträgt $1+(n+1)^d$ bei n Philosophen pro Problem und Rekursionstiefe d ($d,n \in \mathbf{N}, d > 0, n > 1$). Allerdings werden die Netzinstanzen auch recht langsam vom Garbage-Collector weggeräumt, so dass in der Praxis auch bei Rekursionstiefe `2` schnell einige hundert Netzinstanzen auftreten können.

In den folgenden Unterabschnitten werden anhand dieses Netz-

```
;; only mad scientists may choose higher values here
(define number-of-philosophers 2)

(define recursion-depth 2)
```

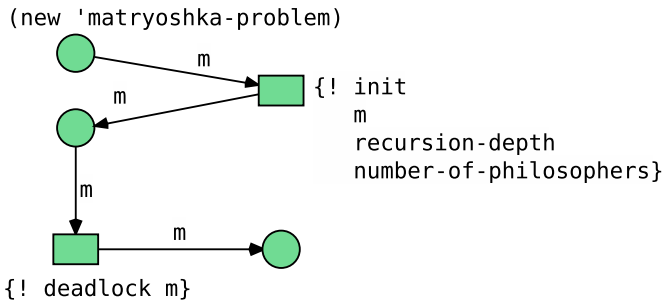

Abbildung 14.3: Russische Philosophen: Netz `matryoshka-init`

systems weitere bereits verwirklichte Möglichkeiten von Scheme-Referenznetzen aufgezeigt.

14.1.2 Programmiersprachliche Abstraktionen

Im Netzsystem der russischen Philosophen werden einige Möglichkeiten der Programmiersprache Scheme genutzt.

- Syntaktische Erweiterung.

  Der neue Bindungsoperator `define-states` erlaubt es, beliebig viele Prozedurdefinitionen mit einem Operator zu erzeugen. Die resultierende Definition ist zwar nicht viel kürzer als die äquivalenten vier, fast identischen, Prozedurdefinitionen, aber weniger redundant und daher weniger fehleranfällig.

- Prozeduren als Objekte erster Klasse und als Marken.

> Die Stelle **`configuration`** enthält eine dynamisch erzeugte Prozedur, die verwendet wird um die Nummer des rechten Essstäbchens zu bestimmen
>
> Des Weiteren wird an einigen Stellen im Netz `matryoshka-problem` die Prozedur *`map`* verwendet, wobei die ordnungserhaltende Eigenschaft von *`map`* nach R^5RS entscheidend ist.

Die Zustandsprozeduren erzeugen darüber hinaus Listen, die Netzexemplare enthalten und die im Deklarationsknoten definierte Prozedur `new-phils` ruft den Operator `new` auf um Netzexemplare zu erzeugen.

Das alles ist zwar grundsätzlich mit geeigneten Methoden auch in Java-Referenznetzen möglich, in diesem Netz werden aber die eingeführten Abstraktionen im Deklarationsknoten explizit sichtbar, was beim derzeitigen Java-Formalismus nicht machbar ist.

Auch die von Scheme bereitgestellten Listenoperatoren wie z. B. *`map`* lassen sich gut an flexiblen Kanten nutzen.

14.1.3 Klar erkennbare Lokalität

Die Auszeichnungen der synchronen Kanäle durch `?` und `!` treten deutlich genug hervor. Zusätzlich wurde bei diesem Netz (und vielen anderen in dieser Arbeit) der von RENEW angebotene `MonoSpace`-Font verwendet, was sich allgemein für Scheme-Petrinetze empfiehlt, um die Einrückung exakt einzuhalten. Im `MonoSpace`-Font sehen auch die geschweiften Klammern deutlich anders aus als die runden, allerdings entsteht allein dadurch keine deutlich sichtbar optische Hervorhebung.

Zusätzlich wurde bei diesem Netz die Einrückung bei synchronen Kanälen so gestaltet, dass alle folgenden Zeilen unter den ersten Kanalparameter eingerückt werden und *nicht* unter den Kanalnamen, so dass der Kanalname dadurch weiter hervorgehoben wird.

Anders als bei Java-Referenznetzen ähnelt sich die Syntax für synchrone Kanäle und für die Erzeugung von Netzexemplaren nicht,

dennoch kann an der gleichen Transition ein Netzexemplar erzeugt und ein Kanal darauf aufgerufen werden (wie z. B. an den Transitionen **`think`** oder **`finish`** im Netz `matryoshka-philosopher`).

Trotzdem ist es sinnvoll, speziell Uplinks grafisch oder durch entsprechende Benennung zusätzlich hervorzuheben. In diesem Netzsystem wurde dies durch die Farbgebung und Namensbestandteile der Transitionen mit Uplinks umgesetzt, sowie dadurch, dass die Uplink-Auszeichnung stets die oberste Beschriftung einer Transition ist.

Für eine Einbettung in Scheme wären ähnliche Konventionen sinnvoll.

14.1.4 Differenzierung: Guard, Bindung, Aktion

Die Bindungsoperatoren `bind` und `is?` wurden von Transitionen so weit abgekoppelt, dass sie auch in benutzerdefinierten Operatoren und auch (eingeschränkt sinnvoll) an Kanten verwendet werden können. Für den (noch nicht implementierten) Operator `action` gilt prinzipiell das gleiche. Auch wenn garantiert sein soll, dass Aktionen erst beim Schalten der Transition nach der Bindungssuche ausgeführt werden, können sie doch an beliebiger Stelle registriert werden, wenn dies sinnvoll erscheint.

Als nicht sinnvoll erscheint dagegen bei einer funktionalen Sprache die Koppelung von Nebeneffekten und Bindung, weswegen die Möglichkeit ausgeschlossen werden soll, Bindungen in Action-Beschriftungen einzuführen.

Dies ist konform mit der Einschränkung, dass der Scheme-Operator *`define`* auch nur am Anfang eines lokalen Gültigkeitsbereichs (vor allen Operationen) verwendet werden darf.

14.1.5 Fluss-, Test-, Reserve- und flexible Kanten

Die gewöhnlichen Kanten, sowie Test- und Reservekanten ließen sich sehr leicht umsetzen und bieten die von Java-Referenznetzen gewohnte Funktionalität. Obwohl die flexiblen Kanten auch nicht

sehr schwer zu implementieren waren, bieten sie zusammen mit den in Scheme verfügbaren Listenoperationen sehr mächtige Werkzeuge.

Im Netzsystem der russischen Philosophen wird über die Beschriftung `(map hungry ps (all n))` an einer flexiblen Kante die Transition **`deadlock`** aktiviert, indem aus der Liste aller Philosophen dynamisch eine Liste der Philosophen mit ihrer Nummer und dem Status `hungry` generiert wird. Derartige Ausdrucksmittel stehen in Java-Referenznetzen nicht direkt zur Verfügung.

Wenn später noch Bibliotheken wie [SRFI-1, 1999] oder [SRFI-42, 2003] im Deklarationsknoten geladen werden können (vgl. Abschnitt 14.2.3 (S. 194)), werden die Ausdrucksmöglichkeiten, auch in Verbindung mit den später zu realisierenden Flush-Kanten, noch weiter erhöht.

14.2 Ungelöste Probleme

Der Ansatz, Nebeneffekte in Scheme-Referenznetzen auszuschließen, funktioniert leider noch nicht ganz. Dieses und andere Probleme hängen auch mit Fehlern in der Scheme-Implementation zusammen, die in zukünftigen Versionen behoben werden können.

Es fehlen bei der Implementation auch noch einige Details im Vergleich zu anderen in RENEW verfügbaren Formalismen, diese können aber einfach nachgerüstet werden. Die hohe Zeitkomplexität der aktuellen Implementation ist teilweise auf eine Schwäche des Entwurfs zurückzuführen, die aber ebenfalls mit einer geplanten Funktionalität der Implementation SISC behoben werden kann.

14.2.1 Nebeneffekte in Scheme-Referenznetzen

Es wurde festgestellt, dass Nebeneffekte der Beschriftungssprache Verletzungen des Lokalitätsprinzips zur Folge haben, darum ist eine Beschriftungssprache ohne sichtbare Nebeneffekte ein wünschenswertes Ziel.

Zwei Mechanismen sollen in Scheme-Referenznetzen ermöglichen, dass Nebeneffekte ausgeschlossen werden können: zum Einen die durch `make-child-environment` mit einer „copy-on-write“ Semantik geschützten Bindungen der globalen Umgebung, zum Anderen die Möglichkeit, Bindungen zu entfernen oder umzudefinieren, so dass keine Bindungen mehr zur Verfügung stehen, die sichtbare Nebeneffekte erzeugen können.

Eine Prozedur `pure-r5rs-subset` soll eine solche Untermenge der R^{5}RS-Standardumgebung erzeugen:

```
(define (pure-r5rs-subset)
  (let ((env (scheme-report-environment 5))
        (disallowed-syntax '(set!)))
    (strict-r5rs-compliance #t)
    (for-each (lambda (proc)
                (eval `(define ,proc
                         (lambda args
                           (,error "forbidden
                               procedure: ~a" ,proc)))
                      env))
              unpure-procedures)
    (for-each (lambda (sym) (disallow-syntax sym env))
              disallowed-syntax)
    (eval ($sc-put-cte-redefinition) env)
    (make-child-environment env)))
```

Hier werden die in SISC verfügbaren modifizierbaren Umgebungen ausgenutzt. Alle unerwünschten Bindungen werden überschattet und anschließend wird die modifizierte Umgebung durch `make-child-environment` geschützt.

Die Liste der unerwünschten Prozeduren ist:

```
(define unpure-procedures
  '(call-with-output-file
    close-input-port
    close-output-port
    interaction-environment
    open-output-file
    null-environment
    scheme-report-environment
    set-car!
```

```
set-cdr!
sisc-initial-environment
string-set!
vector-fill!
vector-set!
with-output-to-file))
```

Wobei hier hauptsächlich Prozeduren auftauchen, die den Zustand von Objekten verändern, aber auch vereinzelt solche, die den Zustand des Systems in einer Weise verändern, der mit anderen Prozeduren sichtbar gemacht werden kann, wie etwa `call-with-output-file` oder `with-output-to-file`. Die Prozedur `scheme-report-environment` ist verboten, weil mit ihr wiederum eine Umgebung erhalten werden könnte, in der Mutationen zur Verfügung stehen.

Der Operator *`set!`* wird mit *`define-syntax`* neu definiert, so dass seine Verwendung einen Fehler erzeugt. Die Prozedur `$sc-put-cte`, welche PSYNTAX zum Einfügen von Bezeichnern in die aktuelle Umgebung verwendet, wird so umdefiniert, dass dies nur einmalig geschehen kann, damit *`define`* auf einen bereits definierten Bezeichner nicht wie *`set!`* wirkt.

Andere Bindungen, die zwar nicht referentiell transparent oder nicht entfaltbar sind[3], aber keine sichtbaren Nebeneffekte erzeugen, wie etwa `quote` oder `read-char` werden unverändert gelassen, da sie keine Verletzung der Lokalität beinhalten.

Zwei Probleme sorgen nun zusammen dafür, dass dieser Ansatz mit der aktuellen SISC-Version nicht uneingeschränkt funktioniert:

1. Interne Definitionen werden oftmals direkt zu so genannten Syntax-Bezeichnern expandiert, wie z. B. `#%letrec` oder `#%define`. Es handelt sich dabei um interne Syntax-Objekte der Implementation SISC.

2. Ein seit der dritten Revision des Scheme-Standards bekanntes Problem ist nie im Standard behoben worden: *`letrec`* im

[3] Terminologie wiederum nach [Søndergaard und Sestoft, 1990]

> Zusammenspiel mit `call-with-current-continuation` wirkt wie *`set!`*.

Zum Ersten:

Die internen Syntax-Bezeichner von SISC sind keine Symbole und können daher auch nicht überschattet werden. Es ist nicht möglich einen Syntax-Transformer oder eine Prozedur an den Namen `#%letrec` zu binden. Trotzdem können diese Bezeichner in Scheme-Code direkt wie Operatoren verwendet werden. Damit wirken sie praktisch wie geschützte Schlüsselwörter, die nicht im Standard [R^{5}RS, 1998] erwähnt werden.

Zum Zweiten:

Am 9. Februar 1988 sandte Alan Bawden eine kurze Mitteilung an die Newsgroup `comp.lang.scheme`, in der demonstriert wird, wie alleine mit `call-with-current-continuation` und *`letrec`* eine veränderbare Zelle erzeugt werden kann ([Bawden, 1988]). Der Code von Alan Bawden nutzt den Umstand aus, dass die Expansion von *`letrec`* oder seine Definition als primitiver Operator zunächst die Bindungen der linken Seiten erzeugt, dann die rechten Seiten an temporäre Variablen gebunden werden und dann eine *Zuweisung* der rechten Seiten an die linke erfolgt. Diese Zuweisung kann erneut durchgeführt werden, indem die Continuation der rechten Seite eingefangen und erneut ausgeführt wird.

Die in [Felleisen u. a., 1991] erarbeitete Lösung lässt sich als Makrodefinition für SISC umsetzen, so dass die rechten Seiten von *`define`* und *`letrec`* Klauseln nur einmalig aufgerufen werden können, aber wenn der ursprüngliche Code von Alan Bawden so umgeschrieben wird, dass er `#%letrec` statt *`letrec`* verwendet, können wieder veränderbare Zellen definiert werden.

Auch spätere Erweiterung dieser Attacken von Al Petrofsky [Petrofsky, 2001] lassen sich durch umdefinieren von *`letrec`* nicht verhindern, weil sie *`define`* in einer lokalen Umgebung verwenden, was wiederum von PSYNTAX direkt zu Klauseln des internen `#%letrec`-Konstrukts expandiert wird.

Aus diesem Grund kann mit dem aktuellen Stand der SISC-

Implementation und den in der Implementation des Scheme-Plugins verfolgten Ansätzen nicht garantiert werden, dass keine sichtbaren Nebeneffekte in Scheme-Referenznetzen mit `call-with-current-continuation` auftreten.

Erschwerend kommt hinzu, dass selbst wenn `call-with-current-continuation` geopfert wird, einfach mit Hilfe einer Closure und dem syntaktischen Bezeichner `#%set` ebenfalls eine veränderliche Zelle definiert werden kann, was sich zur Zeit nicht effektiv verhindern lässt.

Es besteht Grund zu der Hoffnung, dass nebeneffektfreies Scheme in Referenznetzen später noch mit einfachen Mitteln umsetzbar wird. In [Miller, 2007] bestätigt Scott G. Miller die mir persönlich gegenüber geäußerte Absicht, den PSYNTAX-Expander auszutauschen. Der Expander von André van Tonder[4], die Referenzimplementation des Makrosystems für R^{6}RS, soll in Zukunft verwendet werden. Dieser hat die Eigenschaft, dass *`letrec`* und interne Definitionen ausschließlich zu *`lambda`* und *`set!`* expandiert werden, was vermutlich heißt, dass die von Marc Feeley in [Felleisen u. a., 1991] vorgeschlagene Lösung sich damit umsetzen lässt, sofern *`letrec`* nicht weiterhin zum Grundvokabular der Sprache gehört.

Eine aufwändigere Lösung des Problems bestünde darin, den Compiler für Scheme-Referenznetze so abzuwandeln, dass alle Beschriftungen und im Deklarationsknoten geladenen Bibliotheken, zunächst daraufhin analysiert werden, ob verbotene syntaktische Bezeichner direkt verwendet werden.

14.2.2 Unifikation an synchronen Kanälen

Es gibt einen Spezialfall der Unifikation an synchronen Kanälen, der noch nicht funktioniert. Wenn die Ausdrücke auf beiden Seiten

[4] http://www.het.brown.edu/people/andre/macros/index.html

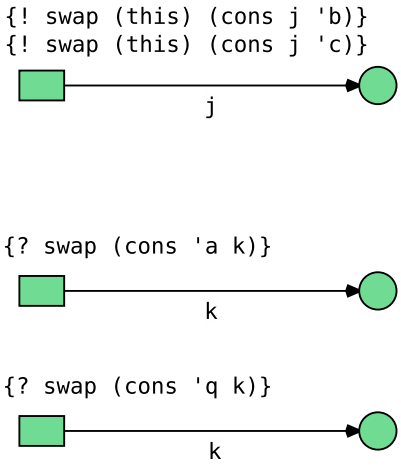

Abbildung 14.4: Kanalparameter mit ungebundenen Variablen auf beiden Seiten

des Kanals Unbekannte enthalten, also echte Unifikation, wie etwa auch bei einem `is?`-Ausdruck, vorgenommen werden soll (siehe Abbildung 14.4), schlägt die Unifikation fehl.

Die Ursache dieses Problems ist noch ungeklärt. Da alle anderen Fälle funktionieren, wäre es aber seltsam wenn gerade dieser Spezialfall auf einen gravierenden konzeptionellen Fehler zurückzuführen wäre. Vermutlich handelt es sich schlicht noch um einen Programmierfehler.

Diese Art der Bindungssuche funktioniert bei Java-Referenznetzen sowohl mit Listen als auch Tupeln (und ist auch ein Beispiel für echte Unifikation im Gegensatz zu bloßem Matching). Wenn die beiden Seiten der „Cons-Zelle“ als einzelne Kanalparameter verwendet werden, funktioniert die Unifikation auch in Scheme-Referenznetzen (vgl. Abbildung 12.4 (S. 173)). Dieses Problem lässt sich also mit einer einfachen Änderung der Notation leicht umgehen, soll aber dennoch später behoben werden.

14.2.3 Probleme der Scheme-Implementation

Die verwendete Version von `psyntax` hat weitere Fehler zur Folge. Das Zusammenspiel von `sisc-initial-environment` und `require-library` ist derart, dass `require-library` nicht mit *`eval`* verwendet werden kann. Damit ist es z. Z. nicht möglich in Scheme-Referenznetzen externe Module zu laden, was ihre praktische Verwendbarkeit etwas einschränkt.

Ich hoffe das dieses Manko ebenfalls mit dem neuen Syntax-Expander behoben werden kann. Andere Scheme-Implementationen (wie z. B. CHICKEN[5]) unterstützen jedenfalls `require-extension` aus [SRFI-55, 2005] auch mit *`eval`*.

Ein anderes Problem besteht darin, dass dynamische Parameter mit *`eval`* doppelt ausgewertet werden, dies konnte aber in der aktuellen Version des Plugins umgangen werden.

Die Änderung des Expanders und andere geplante Änderungen, wie der Austausch des Optimierungsalgorithmus und die Möglichkeit, dynamische Java-Klassen zu generieren, sollten auch wesentlich zu einer besseren Performanz des Scheme-Plugins beitragen (vgl. Abschnitt 14.2.6 (S. 196)).

14.2.4 Inhibitor- und Flush-Kanten sowie „Actions“

In diesem und dem nächsten Unterabschnitt geht es nur noch um Implementationsdetails, die lediglich geringe Priorität hatten und deshalb noch nicht in die aktuelle Version des Scheme-Plugins aufgenommen wurden.

Actions sind zwar wichtig, um Nebeneffekte kontrollieren zu können, wenn sie gewünscht werden, es war aber wesentlich wichtiger zunächst eine funktionierende Umgebung zu schaffen, in der Nebeneffekte erst einmal unerwünscht sind. Ein Netz wie in Abbildung 8.8 (S. 103) kann daher mit der aktuellen Implementation noch nicht funktionieren, da die Java-Methoden eventuell mehrfach

[5] `http://www.call-with-current-continuation.org/`

ausgeführt werden, wenn der `action`-Operator einfach weggelassen wird.

Die bisherigen Erfahrungen bei der Implementation lassen vermuten, dass sich Aktionen relativ einfach hinzufügen lassen, insbesondere da Aktionen in Scheme-Referenznetzen einfacher als in Java-Referenznetzen sind, da sie keine Bindungen modifizieren können. Die Details der Implementation sind aber noch nicht vollkommen geklärt.

Es ist dagegen vollkommen klar, wie Inhibitor- und Clear-Kanten zu implementieren sind, da die dafür nötigen Methodenaufrufe leicht aus den bereits bestehenden Java-Programmteilen abgelesen werden können und die gleichen Konzepte für flexible Kanten und Clear-Kanten anwendbar sein sollten.

Es muss aber auch ein neuer Menüpunkt angezeigt werden, um diese Kanten verfügbar zu machen, und der Simulator muss bei ihrer Verwendung auf den sequentiellen Modus umgeschaltet werden, was Programmieraufgaben sind, die mit dem Konzept der Scheme-Referenznetze wenig zu tun haben.

Aus diesem Grund ist der daraus zu erwartende Erkenntnisgewinn gegenüber dem Programmieraufwand eher gering.

14.2.5 Serialisierung

Scheme-Referenznetze können noch nicht serialisiert werden, was notwendig wäre um mit Shadow-Netzen außerhalb des grafischen Simulators zu arbeiten. Das SISC-Handbuch [Miller und Radestock, 2006] beschreibt sehr detailliert, wie neue Klassen und beliebige Scheme-Objekte serialisiert werden können. Grundsätzlich sind laut Handbuch alle SISC-Objekte serialisierbar (auch Continuations), so dass hier kein grundsätzliches Problem, sondern nur Implementationsarbeit ansteht.

Da die Kommunikation mit Netzen anderer Formalismen (wie z. B. Java-Referenznetzen) nur mit Shadow-Netzsystemen möglich ist, fehlt dieser Teil für einen vollwertigen RENEW-Formalismus.

14.2.6 Performanz

Ein einfaches Scheme-Referenznetz, welches einen Zähler in einer Stelle von 1000 auf 0 herunter zählt und die Gesamtlaufzeit der Simulation misst, läuft z. Z. etwa um den Faktor 30 langsamer als ein äquivalentes Java-Referenznetz. (Netze `performance.rnw` und `performance-java.rnw` im Verzeichnis `Scheme/samples/Regression`.)

Aus diesem Grund habe ich eine einfache Profiling-Library für SISC geschrieben um den Zeitbedarf der einzelnen Prozeduren und Methoden einzuschätzen. Es zeigte sich dabei, dass die meiste Zeit beim Erzeugen von Objekten der Proxy-Hierarchie und von Wrapper-Objekten (siehe Abschnitt 12.3 (S. 143)) verwendet wird.

Die gesamte Hierarchie in Abbildung 12.1 (S. 145) ist lediglich eine Abhilfe dafür, dass Java-Klassen noch nicht direkt in Scheme beerbt werden können. Sie ist aber viel zu allgemein implementiert, beispielsweise werden die Prozeduren zur Implementierung der Proxy-Methoden bei jedem Erzeugen eines Objekts noch dynamisch generiert, was eigentlich nur bei der Methode `get-scheme-wrapper` wirklich nötig wäre.

Die Klassenhierarchie hat den Vorteil, dass sie enorm flexibel ist. Es ist ohne weiteres möglich die gesamte Hierarchie durch Änderungen weniger Zeilen umzustrukturieren, was bei den ersten Experimenten enorm hilfreich war. Daraus erklärt sich auch die Tiefe und Komplexität der Vererbungsbeziehungen.

Diese Flexibilität ist nun nicht mehr nötig und könnte durch ein weniger dynamisches Vorgehen und/oder eine flachere Hierarchie ersetzt werden.

Eine offene Frage ist aber dabei, wie lange eine solche Abhilfe überhaupt noch nötig ist. Wenn in absehbarer Zeit direkt Java-Klassen in SISC-Code beerbt werden könnten, könnte die gesamte Hierarchie gestrichen und durch Java-Klassen ersetzt werden.

Weitere Möglichkeiten zur Optimierung werden in Abschnitt 15.2 (S. 201) angesprochen.

14.3 Auswertung

Sowohl das in dieser Arbeit vorgestellte Konzept als auch die Implementierung des Scheme-Plugins zeigen, dass Scheme als Beschriftungssprache für gefärbte Netze und Referenznetze mindestens ebenso tauglich ist, wie andere Programmiersprachen. Die noch bestehenden Schwächen sind ausschließlich Schwächen der Implementation des Scheme-Plugins, die teils durch Zeitmangel, teils durch noch bestehende Schwächen der Implementation SISC begründet sind.

Die Erweiterungen und Einschränkungen, die gegenüber dem Formalismus der Java-Referenznetze vorgenommen wurden, übertragen das Konzept der Referenznetze angemessen auf die Programmiersprache Scheme.

Die Erweiterungsmöglichkeiten sowohl des Konzeptes als auch der Implementation gehen weit über das jetzt schon offensichtliche hinaus.

Außerdem sollte dieser Text und der erstellte Programmcode hilfreich für die Integration einer weiteren Programmiersprache mit Referenznetzen und/oder RENEW sein.

15 Ausblick

In diesem Kapitel wird näher auf Erweiterungen der Implementation und des Konzepts eingegangen, die im bisherigen Text schon angedeutet wurden.

15.1 Vervollständigung

15.1.1 Scheme-Referenznetze als vollwertiger RENEW-Formalismus

Damit Scheme-Referenznetze in RENEW wirklich alle Möglichkeiten der Java-Referenznetze bieten, fehlt noch die Möglichkeit zur Serialisierung, sowie Clear- (bzw. Flush-) und Inhibitor-Kanten.

Die Kommunikation mit Java-Referenznetzen über synchrone Kanäle kann dann so erfolgen, dass Scheme-Referenznetze einfach Java-Objekte ausliefern bzw. in Empfang nehmen und ihrerseits in Scheme-Objekte umwandeln. Ein Java-Referenznetz sollte für die Kommunikation mit einem Scheme-Referenznetz keine Anpassung benötigen.

15.1.2 Bedienung

Voreinstellungen

Momentan bleiben im Scheme-Formalismus noch alle Voreinstellungen unangetastet. Für Scheme-Referenznetze sollten diese wie folgt angepasst werden:

- Transitionsbeschriftung: Leer, weil in jedem Fall editiert werden muss und spezielle Operatoren und Scheme-Operatoren

vermutlich gleich häufig sind.

- Kantenbeschriftung: kann „x“ bleiben

- Stellenbeschriftung: „()“. SISC akzeptiert zwar die bisherige Voreinstellung [] auch als leere Liste, runde Klammern sind aber als Voreinstellung konform zum Standard.

- Font: MonoSpace, Ausrichtung: linksbündig. Das ist die einzige Kombination, in der komplexere Scheme-Ausdrücke sinnvoll formatiert werden können.

Eine mögliche Erweiterung bestünde noch darin, dass bei der Auswahl des Scheme-Formalismus (im Menü oder mit einer Property) direkt der RENEW-Prompt durch eine Scheme-REPL ersetzt wird. Dazu müsste es allerdings komfortabel möglich sein, aus der REPL heraus wiederum RENEW-Kommandos auszuführen.

Fehlerbehandlung

Sowohl die Behandlung von Laufzeitfehlern während der Simulation als auch Möglichkeiten zur interaktiven Fehlersuche können noch verbessert werden.

Es wäre sinnvoll, Scheme-Laufzeitfehler in Beschriftungen von anderen Laufzeitfehlern zu unterscheiden und erstere im grafischen Simulator in einer Dialogbox anzuzeigen.

Außerdem wäre es sinnvoll wenn bei einem solchen Laufzeitfehler eine REPL in der aktuellen Umgebung gestartet werden könnte. Operatoren zur Inspektion des Stacks und von Objekten in der Umgebung wären ebenfalls hilfreich.

Außerdem sollte die Möglichkeit in der Simulation Breakpoints zu setzen, so erweitert werden, dass in der entsprechenden Umgebung ebenfalls eine REPL gestartet werden kann.

Editor

Momentan muss für komplexere Scheme-Beschriftungen auf einen externen Editor zurückgegriffen werden. Es sollten daher mindestens rudimentäre Hilfen zum Editieren und korrekten Einrücken von Scheme-Code in RENEW geschaffen werden. Optional, aber sehr hilfreich, wäre noch die Möglichkeit, einen Ausdruck aus der Zeichnung zum Testen an die REPL zu senden.

15.2 Optimierung

Die aktuelle Implementation ist schnell genug, um interaktiv Scheme-Referenznetze in RENEW auszuprobieren. Auch lassen sich damit schon interessante Konzepte umsetzen und illustrieren (wie z. B. das Philosophenbeispiel aus Abschnitt 14.1.1 (S. 180)).

Wenn allerdings eine aufwändigere Anwendung als Referenznetzsystem implementiert werden sollte, wäre die derzeitige Performanz gegenüber Java u. U. ein starkes Argument gegen die Verwendung von Scheme.

Die als Nebenprodukt dieser Arbeit erstellte einfache Profiling-Bibliothek für SISC bietet einen Ansatzpunkt, um das Laufzeitverhalten des Plugins besser einschätzen zu können.

Auf Grund der ersten Profiling-Ergebnisse scheint es am vielversprechendsten zunächst die Klassenhierarchie aus Abschnitt 12.3 (S. 143) zu ersetzen oder zu vereinfachen (vgl. auch Abschnitt 14.2.6 (S. 196)).

Sollte das noch nicht ausreichen, besteht auch die Möglichkeit zeitkritische Prozeduren nativ in Java zu implementieren. SISC bietet die dafür notwendige Unterstützung laut [Miller und Radestock, 2006].

Auch die geplanten Änderungen an SISC selbst lassen eine Verbesserung der Performanz erwarten. Sofern es um reine Scheme-Referenznetzsysteme geht, bietet auch eine Einbettung in Scheme u. U. eine performantere Plattform (vgl. auch Abschnitt 7.1.2 (S. 66)).

Weitere Möglichkeiten der Optimierung ergeben sich daraus, dass das SISC System direkten Zugriff auf die Interna des Compilers bietet. Dadurch wird es zumindest denkbar, die Beschriftungen vor dem Start des Simulators in eine effizientere Byte-Code Form zu übersetzen. Dass die sekundäre Umgebung für jede Auswertung neu erzeugt wird, hat den Nachteil, dass solche Übersetzungen und andere Informationen dabei ungültig werden und aufwändig mit `eval` neu erzeugt werden müssen. Dieser Aufwand ließe sich einsparen, wenn Umgebungen in SISC statt dessen in einen definierten Zustand zurückversetzt werden könnten.

15.3 Einbettung in Scheme

Eine sehr interessante Erweiterungsmöglichkeit, die leider im Rahmen dieser Arbeit nur sehr skizzenhaft behandelt werden konnte, ist die u. a. in den Abschnitten 9.3 (S. 116) und 7.1.2 (S. 66) angesprochene Einbettung von Referenznetzen in die Sprache Scheme.

Es wurde bereits festgestellt, dass eine Einbettung in reinem R^5RS-Scheme schwierig wäre, da bereits das Konzept der Scheme-Referenznetze Möglichkeiten benötigt, die der Standard nicht bietet.

Unabhängig davon kann aber zunächst einmal die Syntax von Netzbeschreibungen definiert werden. Die Netzbeschreibungen werden in Netzmuster übersetzt, die ihrerseits zu Netzexemplaren instanziiert werden können. Für beides müssen geeignete, möglichst portable, Objekte verwendet werden.

Dann werden diese Objekte in einem Simulator verwendet, der ähnliche Aufgaben erfüllt wie der RENEW-Simulator, der aber im Gegensatz dazu auf eine portable Bibliothek zur Termunifikation zurückgreifen kann, so dass nicht alle Teile des RENEW-Simulators in Scheme nachimplementiert werden müssten.

Unter Umständen kann die hier vorgeschlagene Semantik der primären und sekundären Umgebungen auch mit anderen Mitteln (wie z. B. geeigneten Modulen) umgesetzt werden.

An einem Punkt der Konzeption sollte klar sein, welche Möglichkeiten eine Scheme-Implementation unbedingt bieten muss, damit eine Einbettung von Scheme-Referenznetzen sinnvoll funktioniert. Die Implementation sollte dann ausschließlich diese Möglichkeiten verwenden.

Ein Im- und Export von grafischen Referenznetzen nach und aus RENEW sollte möglichst früh möglich sein. Eine Idee zur Umsetzung des Imports wäre, dass Netzbeschreibungen in grafische Elemente mit Beschriftungen übersetzt werden und alle anderen Ausdrücke im gleichen Programmtext einfach zum Deklarationsknoten hinzugefügt werden. Umgekehrt würde dann beim Export verfahren.

15.4 Abgeleitete Formalismen und andere Plugins

Das Scheme-Plugin für RENEW lässt sich mit sehr wenig Aufwand so erweitern, dass weitere Plugins in Scheme implementiert werden können. Bereits mit den verfügbaren Mitteln lassen sich schnell Formalismen ableiten, indem einfach die Prozedur überschrieben wird, welche die globale Umgebung erstellt.

Der augenfälligste Kandidat dafür wäre die Untermenge von R^5RS-Scheme ohne Nebeneffekte, sobald sie über einen geeigneten Mechanismus erzwungen werden kann, da sich darüber ganze Netzsysteme ohne Nebeneffekte erstellen ließen, ohne dass jedes Netzmuster die gleiche Deklaration enthalten muss.

Aber auch andere Erweiterungen wären in Scheme denkbar, allerdings sollte dafür erst eine geeignete Schnittstelle definiert und implementiert werden, damit die neuen Plugins einfach auf vorhandene Problemlösungen zurückgreifen können.

15.5 Analyse

Die Vermeidung von Nebeneffekten in Scheme-Referenznetzen war ein Teilziel dieser Arbeit, unter anderem um die spätere Analyse zu erleichtern.

Obwohl Nebeneffekte noch nicht vollständig *verhindert* werden können, können bereits problemlos nebeneffektfreie Scheme-Referenznetze *implementiert* werden. Die meisten Beispiele in dieser Arbeit kommen immerhin ohne sichtbare Nebeneffekte aus. Unter dieser Zusicherung ließen sich also schon Optimierungen vornehmen, die bei Verletzung der Zusicherung eben zu falschen Ergebnissen führen.

Welche Analyseverfahren für Scheme-Referenznetze schon konkret geeignet sind, ist aber noch vollkommen offen.

Abbildungsverzeichnis

Literaturverzeichnis

[Abelson u. a. 1985] ABELSON, H. ; SUSSMAN, G.J. ; SUSSMAN, J.: *Structure and interpretation of computer programs*. MIT Press, 1985 37, 67, 151

[Baldan u. a. 2000] BALDAN, P. ; BUSI, N. ; CORRADINI, A. ; PINNA, G.M.: Functional Concurrent Semantics for Petri Nets with Read and Inhibitor Arcs. In: *CONCUR 2000 - Concurrency Theory: 11th International Conference, University Park, PA, USA, August 2000. Proceedings* (2000), S. 442–457 17, 91

[Bawden 1988] BAWDEN, Alan: *!* Message-ID: <19890302162742.4.ALAN@PIGPEN.AI.MIT.EDU> an die Newsgroup comp.lang.scheme. Februar 1988. – URL http://groups.google.de/group/comp.lang.scheme/msg/b3d8f01faed81302 191

[Craig 1993] CRAIG, Edward: *Analyse? Danke, Nein*. Kap. 1. In: *Was wir wissen können*, Suhrkamp, Frankfurt am Main, 1993 16

[Delgado Friedrichs 2006] DELGADO FRIEDRICHS, Friedrich: *Integration des Model Checking Tools* MARIA *in die Petrinetz Entwicklungsumgebung* RENEW, Universität Hamburg, Department Informatik, Studienarbeit, September 2006 3, 15, 16, 18, 32, 68

[Dybvig 1992] DYBVIG, R.K.: Writing Hygienic Macros in Scheme with Syntax-Case. In: *Computer Science Department, Indiana University, June* (1992) 45, 126, 128

[Felleisen u. a. 1991] FELLEISEN, Matthias ; DUBA, Bruce ; FEELEY, Marc ; HAYNES, Chris ; ROZAS, Guillermo J.: *Declarative LETREC and DEFINE*. Posting an rrrs-authors@mit.edu mit nachfolgender Diskussion. Januar 1991. – URL http://swiss.csail.mit.edu/ftpdir/scheme-mail/HTML/rrrs-1991/msg00000.html 191, 192

[Friedman 1988] FRIEDMAN, DP: Applications of continuations. In: *Tutorial at ACM Symposium on Principles of Programming Languages* (1988). – URL http://www.cs.indiana.edu/~dfried/appcont.pdf 39

[Friedman u. a. 2005] FRIEDMAN, D.P. ; BYRD, W.E. ; KISELYOV, O.: *The reasoned schemer*. MIT Press, 2005 114, 129

[Friedman und Felleisen 1996] FRIEDMAN, D.P. ; FELLEISEN, M.: *The Little Schemer*. MIT Press, 1996 108

[Gibbons 2002] GIBBONS, J.: Chapter 5 Calculating Functional Programs. In: CROLE, Roy (Hrsg.): *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction: International Summer School and Workshop, Oxford, UK, April 10-14, 2000. Revised Lectures*. Oxford, UK : Springer-Verlag, Berlin, Germany, 2002 (Lecture Notes in Computer Science 2297), S. 149–202 33

[Girault und Valk 2003] GIRAULT, C. ; VALK, Rüdiger: *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications*. Springer, 2003 96

[Gosling u. a. 2005] GOSLING, James ; JOY, Bill ; STEELE JR., Guy L. ; BRACHA, Gilad: *Java Language Specification*. 3. Sun Microsystems, Inc., Mai 2005. – URL http://java.sun.com/docs/books/jls/ 6

[Graham 1996] GRAHAM, Paul: *ANSI Common Lisp*. Prentice Hall Englewood Cliffs, NJ, 1996 124

[Hoare 1978] HOARE, Sir Charles Antony R.: Communicating sequential processes. In: *Communications of the ACM* 21 (1978), Nr. 8, S. 666–677 87

[Hughes 1989] HUGHES, J.: Why Functional Programming Matters. In: *The Computer Journal* 32 (1989), Nr. 2, S. 98–107 35

[IEEE Scheme 1991] IEEE Standard for the Scheme Programming Language. (1991). – IEEE Standard 1178-1990 37

[KANREN 2006] FRIEDMAN, Daniel P. ; BYRD, William E. ; KISELYOV, Oleg: *A declarative applicative logic programming system*. Januar 2006. – URL http://kanren.sourceforge.net/. – Software 80, 114, 129

[Kindler und Völzer 1997] KINDLER, Ekkard ; VÖLZER, Hagen: Flexibility in algebraic nets. Humboldt-Universität zu Berlin, November 1997 (89). – Informatik-Berichte. – URL http://www.tcs.uni-luebeck.de/pages/voelzer/Publications/KiVo98.ps 32, 92

[Kiselyov 2002] KISELYOV, Oleg: *How to Write Seemingly Unhygienic and Referentially Opaque Macros with Syntax-rules*. Oktober 2002. – URL citeseer.ist.psu.edu/550415.html 39

[Kristensen und Christensen 2004] KRISTENSEN, Lars M. ; CHRISTENSEN, Søren: Implementing Coloured Petri Nets Using a Functional Programming Language. In: *Higher-Order and Symbolic Computation* 17 (2004), Nr. 3, S. 207–243 14, 17, 23, 25

[Kummer 2002] KUMMER, Olaf: *Referenznetze*. Berlin : Logos Verlag, 2002. – URL http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=deu&id=. – ISBN 3-8325-0035-9 4, 5, 13, 17, 18, 24, 29, 43, 48, 53, 94, 104, 139, 140, 159

[Kummer u. a. 2006a] KUMMER, Olaf ; WIENBERG, Frank ; DUVIGNEAU, Michael: *Renew – Architecture Guide*. Release 2.1. Hamburg: University of Hamburg, Faculty of Informatics, Theoretical Foundations Group (Veranst.), Mai 2006. – URL http://www.renew.de/ 139, 140, 148, 160

[Kummer u. a. 2006b] KUMMER, Olaf ; WIENBERG, Frank ; DUVIGNEAU, Michael: *Renew – User Guide*. Release 2.1. Hamburg: University of Hamburg, Faculty of Informatics, Theoretical Foundations Group (Veranst.), Mai 2006. – URL http://www.renew.de/ 4, 5, 8, 11, 18, 92, 101, 104, 106, 178

[Kummer u. a. 2004] KUMMER, Olaf ; WIENBERG, Frank ; DUVIGNEAU, Michael ; SCHUMACHER, Jörn ; KÖHLER, Michael ; MOLDT, Daniel ; RÖLKE, Heiko ; VALK, Rüdiger: An Extensible Editor and Simulation Engine for Petri Nets: Renew. In: CORTADELLA, Jordi (Hrsg.) ; REISIG, Wolfgang (Hrsg.): *Applications and Theory of Petri Nets 2004. 25th International Conference, ICATPN 2004, Bologna, Italy, June 2004. Proceedings* Bd. 3099. Heidelberg : Springer, Juni 2004, S. 484–493. – URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3099&spage=484 3

[Launchbury 1993] LAUNCHBURY, J.: Lazy Imperative Programming. In: *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIPL '92*, Yale University Research Report YALEU/DCS/RR-968, 1993, S. 46–56. – URL citeseer.ist.psu.edu/launchbury93lazy.html 33, 35

[Mäkelä 2001] MÄKELÄ, Marko: A Reachability Analyser for Algebraic System Nets. Espoo, Finland : Helsinki University of Technology, Laboratory for Theoretical Computer Science, Juni 2001 (A69). – Research Report. – 93 S. – URL http://www.tcs.hut.fi/Publications/info/bibdb.HUT-TCS-A69.shtml 18, 30

[Mäkelä 2002] MÄKELÄ, Marko: Maria: Modular Reachability Analyser for Algebraic System Nets. In: ESPARZA, Javier (Hrsg.) ; LAKOS, Charles (Hrsg.): *Application and Theory of Petri Nets 2002: 23rd International Conference, ICATPN 2002*. Adelaide, Australia : Springer-Verlag, Berlin, Germany, Juni 2002 (Lecture Notes in Computer Science 2360), S. 434–444. – URL http://www.tcs.hut.fi/Publications/msmakela/maria.pdf 24

[Mäkelä 2003] MÄKELÄ, Marko: *Efficient Computer-Aided Verification of Parallel and Distributed Software Systems*. Espoo, Finland, Helsinki University of Technology, Dissertation, November 2003. – URL http://lib.tkk.fi/Diss/2003/isbn9512267926/ 18, 30

[Mäkelä u. a. 2005] MÄKELÄ, Marko ; LATVALA, Timo ; VARPAANIEMI, Kimmo: *Maria 1.3.5—Modular Reachability Analyser*. Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio (Helsinki University of Technology, Laboratory for Theoretical Computer Science), Espoo, Finland. Juli 2005. – URL http://www.tcs.hut.fi/Software/maria/ 32

[Meijer und Finne 2001] MEIJER, E. ; FINNE, S.: Lambada, Haskell as a better java. In: *Electronic Notes in Theoretical Computer Science* 41 (2001), Nr. 1. – URL citeseer.ist.psu.edu/article/meijer01lambada.html 34

[Miller 2002] MILLER, Scott G.: *SISC: A Complete Scheme Interpreter in Java*. Januar 2002. – URL http://citeseer.ist.psu.edu/miller02sisc.html 46, 47

[Miller 2007] MILLER, Scott G.: *Re: [Sisc-users] Large programming project in SISC almost finished*. Message-ID: <a0f746ce0709031939r1df8f258v62e5685e264e63c1@mail.gmail.com> an die Mailing-Liste sisc-users@lists.sourceforge.net. September 2007 128, 192

[Miller und Radestock 2006] MILLER, Scott G. ; RADESTOCK, Matthias: *SISC for Seasoned Schemers*. 2006. – URL http://sisc-scheme.org/manual/html/. – Software Dokumentation 51, 56, 57, 127, 132, 133, 178, 195, 201

[Petrofsky 2001] PETROFSKY, Alopecia: *Obscure continuation hackery combined with non-trivial macrology*. Message-ID: <87u1vrce18.fsf@radish.petrofsky.org> an die Newsgroup comp.lang.scheme. November 2001. – URL http://groups.google.de/group/comp.lang.scheme/msg/bc27beffba1bfdb9 191

[Peyton Jones 2003] PEYTON JONES, Simon L. (Hrsg.): *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003 33

[R^{5}RS 1998] ABELSON, H. ; DYBVIG, R.K. ; HAYNES, C.T. ; ROZAS, G.J. ; ADAMS IV, N.I. ; FRIEDMAN, D.P. ; KOHLBECKER, E. ; STEELE JR., G.L. ; BARTLEY, D.H. ; HALSTEAD, R. ; OXLEY, D. ; SUSSMAN, G.J. ; BROOKS, G. ; HANSON, C. ; PITMAN, K.M. ; WAND, M.: Revised^5 Report on the Algorithmic Language Scheme. In: *Higher-Order and Symbolic Computation* 11 (1998), August, Nr. 1. – URL http://www.schemers.org/Documents/Standards/R5RS/ 35, 37, 38, 39, 45, 56, 59, 84, 88, 164, 178, 191

[Reinke 2000] REINKE, Claus: Haskell-Coloured Petri Nets. In: *IFL '99: Selected Papers from the 11th International Workshop on Implementation of Functional Languages*. London, UK : Springer-Verlag, 2000, S. 165–180. – ISBN 3-540-67864-6 24, 33, 66

[Sabry 1998] SABRY, Amr: What is a Purely Functional Language? In: *Journal of Functional Programming* 8 (1998), Nr. 1, S. 1–22. – URL citeseer.ist.psu.edu/sabry98what.html 31

[SCHELOG 2003] SITARAM, Dorai: *Programming in Schelog*. Juni 2003. – URL http://www.ccs.neu.edu/home/dorai/schelog/schelog.html. – Software und Handbuch, Version 2003-06-01 42, 114

[Schumacher 2003] SCHUMACHER, Jörn: *Eine Plugin-Architektur für Renew – Konzepte, Methoden, Umsetzung*, Universität Hamburg, Department Informatik, Diplomarbeit, 2003 131, 139

[SISC 2006] MILLER, Scott G. ; RADESTOCK, Matthias: *SISC - Second Interpreter of Scheme Code*. 2006. – URL http://sisc-scheme.org/. – Software, Version 1.16.6 29

[SISC Javadoc] MILLER, Scott G. ; RADESTOCK, Matthias: *SISC - API Javadocs*. – URL http://sisc-scheme.org/manual/javadoc/. – Die Dokumentation kann auch aus den aktuellen SISC-Quellen generiert werden. 51, 132, 133

[Søndergaard und Sestoft 1990] SØNDERGAARD, Harald ; SESTOFT, Peter: Referential transparency, definiteness and unfoldability. In: *Acta Informatica* 27 (1990), Nr. 6, S. 505–517. – ISSN 0001-5903 33, 35, 190

[SRFI-1 1999] SHIVERS, Olin: *SRFI 1: List Library*. Oktober 1999. – URL http://srfi.schemers.org/srfi-1/. – Scheme Request For Implementation 125, 188

[SRFI-6 1999] CLINGER, William D.: *SRFI 6: Basic String Ports*. Juli 1999. – URL http://srfi.schemers.org/srfi-6/. – Scheme Request For Implementation 125, 170

[SRFI-8 1999] STONE, John D.: *SRFI 8: `receive`: Binding to multiple values*. August 1999. – URL http://srfi.schemers.org/srfi-8/. – Scheme Request For Implementation 125, 170

[SRFI-9 1999] KELSEY, Richard: *SRFI 9: Defining Record Types*. September 1999. – URL http://srfi.schemers.org/srfi-9/. – Scheme Request For Implementation 62, 125

[SRFI-11 2000] HANSEN, Lars T.: *SRFI 11: Syntax for receiving multiple values*. März 2000. – URL http://srfi.schemers.org/srfi-11/. – Scheme Request For Implementation 125

[SRFI-13 2000] SHIVERS, Olin: *SRFI 13: String Libraries*. Dezember 2000. – URL http://srfi.schemers.org/srfi-13/. – Scheme Request For Implementation 125, 170

[SRFI-39 2003] FEELEY, Marc: *SRFI 39: Parameter objects*. Juni 2003. – URL http://srfi.schemers.org/srfi-39/. – Scheme Request For Implementation 125, 162

[SRFI-40 2004] BEWIG, Philip L.: *SRFI 40: A Library of Streams*. August 2004. – URL http://srfi.schemers.org/srfi-40/. – Scheme Request For Implementation 35

[SRFI-42 2003] EGNER, Sebastian: *SRFI 42: Eager Comprehensions*. Juli 2003. – URL http://srfi.schemers.org/srfi-42/. – Scheme Request For Implementation 94, 188

[SRFI-45 2004] TONDER, André van: *SRFI 45: Primitives for Expressing Iterative Lazy Algorithms*. August 2004. – URL http://srfi.schemers.org/srfi-45/. – Scheme Request For Implementation 35

[SRFI-55 2005] WINKELMANN, Felix L. ; FROST, D.C.: *SRFI 55: `require-extension`*. November 2005. – URL http://srfi.schemers.org/srfi-55/. – Scheme Request For Implementation 128, 194

[SRFI-69 2005] KALLIOKOSKI, Panu: *SRFI 69: Basic Hash Tables*. September 2005. – URL http://srfi.schemers.org/srfi-69/. – Scheme Request For Implementation 126

[Suss:75 1975] SUSSMAN, Gerald J. ; GUY LEWIS STEELE, Jr.: Scheme: An Interpreter for Extended Lambda Calculus. MIT AI Lab, Dezember 1975 (AI Lab Memo AIM-349). – Forschungsbericht. – URL http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-349.pdf 46

[Sussman und Steele 1998] SUSSMAN, G.J. ; STEELE, G.L.: The First Report on Scheme Revisited. In: *Higher-Order and Symbolic Computation* 11 (1998), Nr. 4, S. 399–404 151

[Valk 2003a] VALK, Rüdiger: Essential Features of Petri Nets. In: GIRAULT, C. (Hrsg.) ; VALK, Rüdiger (Hrsg.): *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications*. Springer, 2003, S. 9–28 13, 14

[Valk 2003b] VALK, Rüdiger: Intuitive Models. In: GIRAULT, C. (Hrsg.) ; VALK, Rüdiger (Hrsg.): *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications*. Springer, 2003, S. 29–40 14

[Varhol 1991] VARHOL, Peter D.: ML and Colored Petri Nets for Modeling and Simulation: a Little Language for a Big Job. (ML: Meta Language functional programming language). In: *Dr. Dobbs Journal* 16 (1991), September, Nr. 9, S. 76–81. – NewsletterInfo: 40. – ISSN 1044-789X 23

[Vernet 1998] VERNET, Alessandro: The Jaskell Project. In: *A Diploma Project, Swiss Federal Institute of Technology, February* (1998) 34

[Yhc 2006] „YHC TEAM“: *Yhc*. November 2006. – URL http://haskell.org/haskellwiki/Yhc. – Software 34

Erklärungen

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Hamburg, den 27. September 2007
Friedrich Delgado Friedrichs

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Department Informatik einverstanden.

Hamburg, den 27. September 2007
Friedrich Delgado Friedrichs