Objektorientierte Erweiterungen einer deklarativen Programmiersprache

von
Philipp Niederau
Matrikelnummer 205169

Diplomarbeit im Fach Informatik

vorgelegt der
Fakultät für Mathematik, Informatik und Naturwissenschaften
der Rheinisch-Westfälischen Technischen Hochschule Aachen
im August 2000

angefertigt am
Lehr- und Forschungsgebiet für Informatik (I2)
bei
Prof. Dr. Michael Hanus

Ich danke Herrn Prof. Dr. M. Hanus für die Überlassung des Themas und für die Bereitschaft, die vorliegende Arbeit zu begutachten, Herrn Prof. Dr. K. Indermark dafür, dass er sich als Zweitgutachter zur Verfügung gestellt hat, Frank Huch für seine konstruktive Kritik und zahlreichen Diskussionen über das Typsystem sowie meiner Verlobten Johanna für ihre große Unterstützung.

Die vorliegende Arbeit folgt den Regeln der reformierten Rechtschreibung.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 31. August 2000

(Philipp Niederau)

Inhaltsverzeichnis

Einleitung

Deklarative Programmiersprachen ermöglichen gegenüber klassischen imperativen Programmiersprachen die Programmierung auf einem höheren Abstraktionsniveau: Deklarative Programme beschreiben nicht, *wie* eine Aufgabenstellung zu lösen ist, sondern geben dem Computer in Form von Regeln und Wissen an, *was* es zu lösen gilt. Die Folge sind bessere Lesbarkeit, Wartbarkeit und kürzere Entwicklungszeiten. Darüber hinaus bieten sie eine Vielzahl von Mechanismen zur Wiederverwendung von Code, insbesondere durch parametrisch polymorphe Typen. Deklarative Programmiersprachen lassen sich in funktionale und logische Sprachen klassifizieren. Während funktionale Sprachen wie Lisp, ML oder Haskell auf dem Lambda-Kalkül basieren, verwenden Logiksprachen den Prädikatenkalkül bzw. Hornklauseln.

Seit einiger Zeit existieren funktional-logische Sprachen, die beide Ansätze kombinieren. In diesem Zusammenhang entstand auch Curry, eine deklarative Multiparadigmensprache. Curry enthält funktionale, logische und nebenläufige Aspekte, wie verschachtelte Ausdrücke, verzögerte Auswertung, Funktionen höherer Ordnung, logische Variablen, partielle Datenstrukturen, eingekapselte Suche und nebenläufige Auswertung von Ausdrücken. Aus der Kombination der verschiedenen Ansätze ergeben sich darüber hinaus neue Programmiertechniken.

Bei der Erstellung von Anwendungen mit einem hohen Anteil an Interaktion, z. B. bei graphischen Benutzeroberflächen oder verteilten Systemen, haben sich in der Praxis objektorientierte Programmiersprachen bewährt. Objekte eignen sich gut zur Modellierung von Gegenständen und Beziehungen der realen Welt. Auch bieten objektorientierte Programmiersprachen durch Vererbung und Subtyppolymorphismus zusätzliche Möglichkeiten bei der Wiederverwendung von Code.

Ziel der Arbeit ist die Entwicklung einer konservativen Erweiterung der Programmiersprache Curry, um sowohl die Stärken der deklarativen als auch die der objektorientierten Programmierung einsetzen zu können. Das Resultat ist die Sprache ObjectCurry, in der zusätzliche Syntax die Definition von Klassen erlaubt, über spezielle Constraints Objekte erzeugt und Nachrichten versandt werden können, sowie mittels eines modifizierten Typsystems ein Höchstmaß an Flexibilität und Typsicherheit erreicht wird.

Kapitel 1 behandelt die Grundlagen objektorientierter Programmiersprachen. Insbesondere werden wir einige Merkmale aufführen, die eine Programmiersprache aufweisen muss, um sich *objektorientiert* nennen zu dürfen. Anschließend stellen wir die deklarative Programmiersprache Curry vor, die eine zentrale Rolle in dieser Arbeit spielt. Es folgt eine Diskussion über Vor- und Nachteile deklarativer bzw. objektorientierter Programmierung und wir geben einen Überblick über existierende objektorientierte Erweiterungen einiger deklarativer Programmiersprachen.

In Kapitel 2 werden wir sehen, dass auch in Curry objektorientierte Programmierung[1] durch den Einsatz einer bestimmten Programmiertechnik möglich ist.

Jede Programmiertechnik birgt jedoch gewisse Nachteile, denen durch eine Erweiterung der Programmiersprache begegnet werden kann. Eine solche Erweiterung wird in Form der Programmiersprache ObjectCurry in Kapitel 3 vorgestellt. In dieser werden außerdem weitere Konzepte realisiert wie Vererbung und Subtyppolymorphismus, die in Kapitel 2 außer Acht gelassen werden. Ziel bei der Entwicklung von ObjectCurry war es, möglichst alle Merkmale der Objektorientierung, wie sie in Kapitel 1 eingeführt werden, zu integrieren und so *echte* objektorientierte Programmierung in Curry zu ermöglichen, gleichzeitig aber die Vorzüge von Curry beizubehalten.

Kapitel 4 beschreibt die Möglichkeiten des Einsatzes von ObjectCurry im Bereich der verteilten Programmierung. Um dieses wichtige Anwendungsgebiet weiter zu unterstützen, integrieren wir zusätzliche Mechanismen zur Synchronisation von Nachrichten nach dem Monitor-Konzept in die Sprache.

Da objektorientierte Programmiersprachen über ein Typsystem mit Subtyppolymorphismus verfügen, das von Curry jedoch über parametrischen Polymorphismus, und beide Konzepte orthogonal zueinander liegen, ist es notwendig, ObjectCurry ein neues Typsystem zu geben. Dieses wird in Kapitel 5 ausführlich behandelt. Dabei betrachten wir zunächst die unterschiedlichen Formen von Polymorphismus und stellen das Hindley-Milner-ähnliche Typsystem von Curry vor. Im Anschluss daran wird das Typsystem von ObjectCurry formal definiert und erläutert. Den Abschluss bildet die Vorstellung eines Algorithmus zur Typinferenz.

Als praktisch ausgerichtete Diplomarbeit beinhaltet diese auch einen Compiler und einen Typchecker für die Sprache ObjectCurry. Kapitel 6 geht konkret auf den ObjectCurry-Compiler, den ObjectCurry-Typchecker und die Implementierung dieser Module ein. Im Rahmen des Compilers wurde zudem ein Parser für Curry in Curry auf der Grundlage von Parsergeneratoren entwickelt, der sehr gut für zukünftige Erweiterungen von Curry bzw. ObjectCurry eingesetzt werden kann. Das Konzept der Parsergeneratoren wurde zu diesem Zweck um Operatoren zur Behandlung der in Curry und ObjectCurry verwendeten Offside-Rule erweitert.

Die Arbeit schließt mit einer Zusammenstellung der erzielten Ergebnisse und offen gebliebenen Fragen.

[1] ohne Vererbung

1. Grundlagen

1.1. Objektorientierte Programmierung

Objektorientierte Programmierung bedeutet Programmierung nach dem Konzept der **Objektorientierung** [JCJÖ97, S. 84]. Objektorientierung spielt in der Industrie ohne Frage eine große Rolle im Bereich der Softwareentwicklung. Objektorientierung findet sich nicht nur im Bereich der Programmiersprachen, sondern auch in anderen Phasen der Softwareentwicklung, so beispielsweise in verschiedenen Analyse- und Modellierungstechniken, in Testverfahren oder in objektorientierten Datenbanken.

Das folgende Kapitel stellt die grundlegenden Prinzipien und Begriffe der Objektorientierung vor. Im Anschluss daran werden wir sehen, wie sich diese in objektorientierten Programmiersprachen wiederfinden.

1.1.1. Was ist Objektorientierung?

In Jacobson et al. [JCJÖ97] finden wir: „Objektorientierung ist eine Technik zur Modellierung eines Systems. Sie beinhaltet eine Reihe von zu diesem Zweck geeigneten Konzepten. Das Wort ‚System' ist hierbei weit gefasst und kann sowohl als Softwaresystem, als auch als System in einem größeren Kontext aufgefasst werden [...]".

Als Vorteile von objektorientierten Systemen sieht Jacobson:

1. das einfachere Verständnis, da der semantische Graben zwischen Modell und Wirklichkeit klein gehalten wird
2. die Lokalität von Modifikationen

Insbesondere der erste Punkt wird jedoch auch von den Befürwortern deklarativer Programmiersprachen aufgeführt. Wir werden daher noch genauer untersuchen müssen, worin die Stärken der Objektorientierung bestehen.
Um zu verstehen, was Objektorientierung bedeutet, muss zunächst auf deren wesentliche Basiskonzepte und Begriffe eingegangen werden. Definitionen und Beschreibungen objektorientierter Konzepte finden sich in vielen Variationen in der Literatur. Wir beschränken uns an dieser Stelle auf einen kurzen Überblick.

Zentraler Gegenstand jedes objektorientierten Systems sind Objekte.

Def. 1.1.1 ***Objekte*** *sind gleichberechtigte, aktiv handelnde Einheiten. Objekte* ***kommunizieren*** *miteinander, indem sie Nachrichten senden und empfangen. Unter einer* ***Nachricht***[1] *versteht man dabei die Aufforderung an ein Objekt, eine bestimmte Aufgabe auszuführen. (nach [W^{+}95, S. 330])*

[1] andere Autoren sprechen statt von Nachrichten auch von **Botschaften**

Objekte **reagieren** auf Nachrichten, die sie erhalten, indem sie **Operationen** [2] ausführen. Diese können beispielsweise darin bestehen, weitere Nachrichten an andere Objekte zu schicken.

Alle Informationen eines rein objektorientierten Systems sind innerhalb der Objekte gespeichert. Die in einem Objekt abgelegten Daten bilden den **Zustand** eines Objekts. Sie lassen sich von außen nicht verändern. Daher spricht man bei Objekten auch von **Datenkapseln**. Manipuliert werden können die Daten nur indirekt durch das Senden einer Nachricht: Als Reaktion auf eine solche Botschaft führt das Objekt Operationen aus, die die internen Daten verändern. Dadurch wechselt das Objekt seinen Zustand.

Objekte besitzen eine **Identität**, d. h. ein Objekt ist nicht allein über seinen Zustand bestimmt. Es können zwei oder mehr verschiedene Objekte mit dem gleichen Zustand zur selben Zeit innerhalb eines Programms existieren. Ihre Identität unterscheidet Objekte von Elementen eines abstrakten Datentyps.

In einem objektorientierten System existieren in der Regel eine Vielzahl von miteinander kommunizierenden Objekten. Einige dieser Objekte besitzen gemeinsame Charakteristika, z. B. reagieren sie in ähnlicher Weise auf Nachrichten. Solche Objekte lassen sich zu einer Gruppe zusammenfassen. Eine Schablone, die die gemeinsamen Charakteristika von Objekten festlegt, heißt im objektorientierten Sprachgebrauch Klasse.

Def. 1.1.2 *Eine **Klasse** ist ein Schema zum Beschreiben von Objekten ähnlichen Verhaltens. Zu einer Klasse können beliebig viele Instanzen, Objekte genannt, erzeugt werden [W+95, S. 347].*

Objekte einer Klasse besitzen also gemeinsame Eigenschaften, die durch die Klasse festgelegt sind. Dazu zählen insbesondere die Nachrichten, auf die die Instanzen der Klasse reagieren, und die Struktur der Informationen, die in den Objekten abgelegt sind. Der aktuelle Zustand einer Instanz ist durch die Operationen bestimmt, die das Objekt bisher ausgeführt hat.

Ein weiterer wichtiger Begriff innerhalb objektorientierter Systeme ist der der **Vererbung**. Ähnlich wie Objekte verfügen mitunter auch Klassen über gemeinsame Eigenschaften. Diese Gemeinsamkeiten lassen sich extrahieren und in einer eigenen Klasse zusammenstellen. Die ursprünglichen Klassen können dann diese Merkmale von der neuen Klasse **erben** ([JCJÖ97, S. 56]). Durch Vererbung bilden Klassen eine Hierarchie. Wenn eine Klasse B von einer Klasse A erbt, so heißt die Klasse A **Vorfahre** oder **Oberklasse** von B, und B heißt **Kind** oder **Unterklasse** von A. Falls A ebenfalls von einer weiteren Klasse X erbt, so wird auch diese Klasse X mitunter als Vorfahre von B bezeichnet. Wir wollen immer von dem **direkten Vorfahren** sprechen, wenn wir die Klasse meinen, von der unmittelbar geerbt wird.

Mehrfachvererbung bedeutet, dass einer Klasse mehrere Oberklassen zugeordnet werden. Bei einigen Problemstellungen kann Mehrfachverbung durchaus sinnvoll sein und die Modellierung erleichtern. Jedoch führt Mehrfachvererbung in der Praxis zu einer Reihe von Problemen (s. Kapitel 3.2). Oftmals wird sie auch in einer Weise eingesetzt, die der

[2]synonym für Operation wird bisweilen auch der Begriff der Methode verwendet

Struktur eines Systems mehr schadet als nützt. Aus diesen Gründen ist Mehrfachvererbung sehr umstritten. Wir werden uns in dieser Arbeit mit Merhfachvererbung daher nicht ausführlich beschäftigen.

Wie in funktionalen Sprachen spielt auch in objektorientierten Systemen der Begriff des **Polymorphismus** eine wichtige Rolle. Es gibt viele Formen von Polymorphismus, und einige davon existieren in nahezu jeder Programmiersprache. Jedoch sind die verschiedenen Arten von Polymorphismus nicht äquivalent, d. h. eine Form kann nicht durch eine andere Form ausgedrückt oder in diese transformiert werden. Die unterschiedlichen Arten von Polymorphismus werden ausführlich in Kapitel 5.1 behandelt.

1.1.2. Objektorientierte Programmiersprachen

Eine objektorientierte Programmiersprache sollte zumindest alle Merkmale der Objektorientierung aufweisen, so dass der Programmierer diese in ihrer Gesamtheit nutzen kann ([JCJÖ97, S. 84]). Daher muss eine objektorientierte Programmiersprache folgende Konzepte beinhalten:

1. gekapselte Objekte
2. Klassen und Instanzen
3. Vererbung zwischen Klassen
4. Polymorphismus

Programmiersprachen, die nur die beiden ersten Merkmale besitzen, werden auch **objektbasiert** genannt.

Das wichtigste Konzept, über das eine objektorientierte bzw. objektbasierte Programmiersprache verfügen muss, ist das des Objekts. Objekte speichern Informationen, kapseln diese von der Außenwelt ab und reagieren auf Nachrichten. Ein Objekt wird in der Regel als Instanz einer Klasse erzeugt. Daher ist die Struktur eines Objekts durch seine Klasse beschrieben. Eine Klasse definiert die **Nachrichten**, die ein Objekt versteht, und die **Reaktionen** auf den Erhalt einer solchen Nachricht. Die gewünschten Reaktionen von Instanzen einer Klasse werden durch die Angabe von **Operationen** festgelegt. Eine mit diesen Operationen assoziierte Nachricht wird **Methode** genannt. Methoden übernehmen in objektorientierten Programmiersprachen die Rolle von Prozeduren und Funktionen aus prozeduralen Sprachen. Des Weiteren legt eine Klasse die **Instanzvariablen** oder **Attribute** eines Objekts fest. Instanzvariablen nehmen die Informationen auf, die in einem Objekt gespeichert werden sollen. Die Werte der Instanzvariablen bilden den aktuellen Zustand eines Objekts. Das kartesische Produkt aller möglichen Werte der einzelnen Instanzvariablen spannt den gesamten Zustandsraum auf, der einem Objekt maximal zur Verfügung steht.

Auf folgende Weisen kann ein Objekt auf eine eingegangene Nachricht reagieren:

- Das Objekt wechselt seinen Zustand, indem es die Belegung seiner Instanzvariablen ändert.
- Das Objekt versendet Nachrichten an andere Objekte.

- Das Objekt liefert eine Antwort an den Sender der Nachricht zurück.

In vielen objektorientierten Sprachen wird das Versenden von Nachrichten und Zurückgeben von Antworten durch den Aufruf von Prozeduren- bzw. Funktionen realisiert. Der Name der Nachricht entspricht dem Namen einer Prozedur oder Funktion, und das Funktionsergebnis ist die Antwort.

Objekte sind in objektorientierten Programmiersprachen Bürger erster Klasse: Sie können in Nachrichten verpackt an andere Objekte verschickt oder als Antworten zurückgegeben werden.

Es existieren mittlerweile eine Vielzahl objektorientierter Programmiersprachen. Einige davon wurden als objektorientierte Sprachen entworfen, andere sind durch die Erweiterung bereits existierender Sprachen entstanden.

Eine andere Unterteilung ist die in rein objektorientierte Programmiersprachen und Hybridsprachen: Bei den meisten Sprachen, die durch eine Erweiterung bestehender Sprachen entstanden sind, handelt es sich um **Hybridsprachen**. In einer solchen existieren die Konzepte der alten Sprache und die der Objektorientierung nebeneinander, so z. B. in C++, Ada oder ObjectPascal. Auf der einen Seite kann die Vermischung unterschiedlicher Paradigmen zu unübersichtlichem Code und schlechtem Programmierstil führen. Andererseits bringt sie aber auch Vorteile, z. B. wenn es um die Darstellung von Werten in objektorientierten Sprachen geht, wie wir in Abschnitt 1.3.1 sehen werden.

Ein Vertreter rein objektorientierter Programmiersprachen ist die Sprache Smalltalk. Hier besteht alles aus Objekten und Nachrichten. Auch Zahlen sind in Smalltalk als Objekte realisiert, an die Nachrichten wie `+` oder `=` geschickt werden können. Klassen sind in Smalltalk ebenfalls Objekte und damit Bürger erster Klasse.

Java ist eine rein objektorientierte Programmiersprache, bietet aber spezielle Ausdrücke und Operationen zur Behandlung von numerischen Ausdrücken an.

1.2. Die Multiparadigmensprache Curry

Curry ist eine deklarative Multiparadigmensprache, denn sie verbindet die Konzepte funktionaler, logischer und nebenläufiger Programmiersprachen. Aus syntaktischer Sicht ist Curry stark an die funktionale Programmiersprache Haskell [P^+97] angelehnt. Ein Curryprogramm besteht aus einer Menge von Datentyp- und Funktionsdefinitionen. Prädikate werden in Curry als Funktionen mit dem vordefinierten Typ `Success` notiert. Funktionsdefinitionen können freie logische Variablen innerhalb ihrer Bedingungen und rechten Regelseiten enthalten. Auch der Aufruf einer Funktion mit nicht instantiierten logischen Variablen als Argumenten ist wie in der Logikprogrammierung möglich. Verzichtet werden konnte in Curry auf sämtliche nicht-deklarativen Features, wie sie beispielsweise Prolog (s. [Han87]) aufweist.

Im Folgenden sollen die Eigenschaften und Merkmale von Curry informell vorgestellt werden, die von besonderer Bedeutung für die folgenden Kapitel sind. Eine vollständige Beschreibung der Sprache findet sich in [Han00a].

1.2.1. Datentypen und Datenterme

Abstrakte Datentypen spielen eine zentrale Rolle bei der Darstellung von Informationen in Curry. Ein abstrakter Datentyp kann über das reservierte Schlüsselwort `data` definiert werden. Die Definition eines n-stelligen Datentyps t hat das folgende Aussehen[3]:

$$\texttt{data}\ t\ a_1\ \ldots\ a_n\ =\ C_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}\ |\ \ldots\ |\ C_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$$

Dabei sind $\tau_{i,j}$ beliebige Typausdrücke. Ein **Typausdruck** ist entweder eine **Typvariable**, ein Datentypkonstruktor oder die Applikation eines Datentypkonstruktors auf weitere Typausdrücke. Der Datentyp t wird aus k **Datentypkonstruktoren** C_i gebildet. Jeder der Datentypkonstruktoren C_i erhält den Typ

$$\tau_{i,1}\ \rightarrow\ \ldots\ \rightarrow\ \tau_{i,n_i}\ \rightarrow\ \tau\ a_1 \ldots\ a_n$$

Die Applikation von C_i auf n_i Elemente konstruiert daher ein Element des Typs t.

Der Name t des so erzeugten Datentyps wird im Typsystem benötigt, taucht jedoch außerhalb von Typausdrücken nicht auf.

Der Datentyp t ist polymorph über den Typvariablen $a_1\ \ldots\ a_n$, d. h. an die Stelle der a_i kann jeder beliebige andere Datentyp treten. Diese Art von Polymorphismus über Typen wird **parametrischer Polymorphismus** genannt. Parametrischer Polymorphismus erhöht die Flexibilität einer Programmiersprache erheblich; so ist es z. B. sehr einfach, Listen über beliebigen Typen zu definieren:

```
data List a = Empty | Cons a (List a)
```

Eine Liste ist entweder leer, oder sie besteht aus einem Element und einer weiteren Liste.

Ein **Datenterm** ist die Anwendung eines n-stelligen Datentypkonstruktors C auf n weitere Datenterme.

1.2.2. Funktionsdefinitionen

Eine Funktion wird in Curry durch eine Typdeklaration und eine Liste von definierenden Gleichungen definiert. Die **Typdeklaration** einer n-stelligen Funktion hat die Form

$$f{::}\tau_1\texttt{->}\tau_2\texttt{->}\cdots\texttt{->}\tau_n\texttt{->}\tau$$

$\tau_1, \ldots, \tau_n, \tau$ sind Typausdrücke und τ ist ein nicht-funktionaler Typ.

Die Form einer **definierenden Gleichung** für eine n-stellige Funktion ($n \geq 0$) ist

$$f t_1 \ldots t_n | c = e$$

[3] In Curry spielt im Gegensatz zu Haskell die Groß- bzw. Kleinschreibung von Bezeichnern keine Rolle.

Dabei sind $t_1, \ldots, t_n$ Datenterme, e ist ein beliebiger Ausdruck und c ein Bedingung, die auch fehlen kann. Eine Bedingung ist entweder ein boolscher Ausdruck oder ein **Constraint** (s. 1.2.4). Die Typdeklaration kann fehlen. In diesem Fall wird der Typ der Funktion durch den Typchecker inferiert.

Ein **Ausdruck** ist eine Variable oder eine (partielle) Applikation $f\ e_1 \ldots e_m$, falls f eine n-stellige Funktion oder ein n-stelliger Datentypkonstruktor, $m \leq n$ und $e_1, \ldots, e_m$ Ausdrücke sind. Ein **Operator** ist eine zweistellige Funktion. Curry erlaubt bei Operatoren die Infixschreibweise.

Auf der linken Seite einer definierenden Gleichung können nicht nur Variablen, sondern beliebige Grundterme auftreten. Diese fungieren als Muster (engl. *pattern*). Eine Regel kann angewendet werden, wenn in einer Funktionsapplikation die Muster der linken Seite auf die Argumente passen. Auf der **linken Seite** (*left-hand side*) $f \tau_1 \ldots \tau_n$ darf keine Variable mehrfach auftreten. Anstelle eines Grundterms kann auch die **anonyme Variable** _ stehen. Diese passt auf jeden Wert. Die Definition von Funktionen durch die Angabe von Mustern wird **Pattern-Matching** genannt.

Die Muster der linken Seiten mehrerer Gleichungen zu einer Funktion können sich überlappen: Zwei Muster überlappen sich, wenn ein Ausdruck existiert, der auf beide Muster passt. In diesem Fall ist die definierte Funktion **nicht-deterministisch**. Die Anwendung einer nicht-deterministischen Funktion auf einen Ausdruck kann zu verschiedenen Lösungen führen. Mit Hilfe der so genannten **eingekapselten Suche** lässt sich in Curry die Suche nach diesen Lösungen steuern. Wir benötigen im Verlauf der Arbeit lediglich den Ausdruck `findfirst (\x -> x =:= e)`, der die erste Lösung des möglicherweise nicht-deterministischen Ausdrucks `e` berechnet. Für Details sei an dieser Stelle auf [Han00a] verwiesen.

1.2.3. Freie Variablen

Curry ermöglicht die Verwendung freier Variablen innerhalb von Ausdrücken und Constraints. Enthält ein Ausdruck eine freie Variable, so kann dieser reduziert werden, wenn die Variable an einen Wert gebunden wird. Jede freie Variable muss deklariert werden. Dies geschieht durch eine Deklaration der Form x `free` innerhalb lokaler Definitionen durch `let` oder `where`. Auch in einem Funktionsaufruf können freie Variablen als Argumente angegeben werden. Über die Angabe einer **Auswertungsstrategie** kann für jede Funktion einzeln festgelegt werden, wie solche Argumente behandelt werden. Curry kennt drei Auswertungsannotationen: `flex`, `rigid` und `choice`.

Eine Funktion mit der Auswertungsstrategie `flex` bindet freie Variablen, die als Argumente übergeben werden.

Bsp. 1.2.1

```
test eval flex
test True = 42
```

Der Ausdruck `test x` *wird zu* `42` *ausgewertet, und* `x` *an* `True` *gebunden.*

Anders verhält es sich, wenn die Funktion `test` mit `rigid` annotiert wird: In diesem Fall wird die Auswertung des Ausdrucks `test x` solange angehalten, bis `x` in einer anderen Berechnung an einen Wert gebunden wird.

Als dritte Auswertungsannotation existiert noch die nicht-deterministische Auswahl `choice`. Für nähere Erläuterungen sei an dieser Stelle auf [Han00a] verwiesen.

1.2.4. Constraints

Definierende Regeln können Constraints beinhalten. Diese Constraints sind zu lösen, bevor die Regel angewendet werden kann. Constraints sind Ausdrücke des vordefinierten Typs `Success`. Der einfachste Constraint heißt `success` und ist immer erfüllt. Eine weitere Gruppe von Constraints sind die Gleichheitsconstraints. Ein **Gleichheitsconstraint** $e_1 =:= e_2$ ist erfüllt, falls beide Seiten auf denselben Grundterm reduziert werden können. Der Gleichheitsconstraint wird strikt gelöst. Falls einer der beiden Ausdrücke undefiniert ist, da seine Berechnung nicht terminiert, hält auch die strikte Gleichheit nicht.

Zwei Constraints c_1 und c_2 können mit Hilfe der Konjunktion zu einem neuen Constraint $c_1 \& c_2$ verbunden werden. Der Constraint $c_1 \& c_2$ ist erfüllt, wenn die Constraints c_1 und c_2 nebenläufig erfüllbar sind.

Die nebenläufige Auswertung kann durch eine serielle Auswertung ersetzt werden, indem statt `&`, der Operator `&>` verwendet wird. Dieser ist wie folgt vordefiniert:

```
(&>) :: Success -> Success -> Success
c1 &> c2 | c1 = c2
```

Bsp. 1.2.2

```
test eval rigid
test x | x =:= y & y =:= True = 42 where y free
```

Wird der Ausdruck `test x` *ausgewertet, so wird zunächst versucht,* `x` *mit* `y` *zu unifizieren. Da sowohl* `x` *als auch* `y` *freie Variablen sind, gelingt dies nicht. Daher wird der Constraint* `x =:= y` *zunächst suspendiert. Der nächste Constraint bindet dann* `y` *an* `True`*. Nun kann der suspendierte Constraint aufgeweckt und gelöst werden, und die Funktion kann zu* `42` *reduziert werden.*

Fehlt die Angabe einer Auswertungsannotation, so sind Constraints als `flex` vordefiniert, alle übrigen Funktionen hingegen als `rigid`. Ein Gleichheitsconstraint bindet daher enthaltene freie Variablen, falls dies möglich ist, und suspendiert nur, wenn eine freie Variable mit einer freien Variable unifiziert wird.

1.2.5. Funktionen höherer Ordnung

Curry unterstützt wie Haskell Funktionen höherer Ordnung, partielle Applikation und Lambda-Abstraktion nach den gängigen funktionalen Techniken. Funktionen werden verzögert ausgewertet und ermöglichen so den Umgang mit unendlichen Datentypen.

1.3. Deklarativ vs. Objektorientiert

Deklarative Programmiersprachen erlauben die Programmierung auf einem hohen Abstraktionsniveau und befreien den Programmierer von viel überflüssiger Schreibarbeit. Objektorientierte Sprachen hingegen eignen sich besonders zur Modellierung von Dingen und Beziehungen der realen Welt. Der folgende Abschnitt vergleicht beide Ansätze und geht auf Stärken und Schwächen ein.

1.3.1. Werte in objektorientierten Programmiersprachen

Klassen und Objekte eignen sich hervorragend zur Modellierung komplexer Strukturen. Daneben bieten objektorientierte Programmiersprachen aber meist nur wenig Unterstützung bei der Strukturierung von Daten, da das Gesamtkonzept möglichst einfach gehalten werden soll („alles ist ein Objekt“). Als Konsequenz hat jedes Datenelement die Eigenschaften, die sich aus dem objektorientierten Ansatz ergeben, nämlich insbesondere einen Zustand und eine Identität. Für gewisse Daten ist dieser Ansatz jedoch nicht angemessen. So ist es beispielsweise fragwürdig, die Zahl 7 zu erzeugen oder mehrere 7-Objekte mit unterschiedlicher Identität zu besitzen. Zahlen sind mathematisch gesehen zeitlose, abstrakte Entitäten. Gleiches gilt für weitere Strukturen wie Vektoren, Listen, Funktionen oder algebraische Datentypen (vgl. [Nor99b, S. 10]).

Für Zahlen stellen die meisten objektorientierten Programmiersprachen daher zusätzliche Mechanismen und Notationen zur Verfügung, so dass diese nicht als Objekte erzeugt werden müssen. Anders verhält es sich mit den übrigen angesprochenen Datentypen. Wird beispielsweise ein Vektor von Objekten benötigt, so muss dieser zunächst als Objekt erzeugt werden und dann mit den einzelnen Komponenten sukzessive gefüllt werden.

Abhilfe schaffen Erweiterungen wie beispielsweise **Pizza** [OW97]. Pizza ist eine Erweiterung von Java um algebraische Datentypen, Pattern-Matching und Funktionen höherer Ordnung.

1.3.2. Interaktion in funktionalen Programmiersprachen

Im Gegensatz zu objektorientierten Programmiersprachen bieten deklarative und insbesondere funktionale Programmiersprachen einen direkten mathematischen Zugang zu Werten. Es existieren einfache Möglichkeiten, einen Wert als Ausdruck zu notieren, zu analysieren oder zu vereinfachen.

Rein funktionale Programmiersprachen bieten jedoch wenig, um Interaktion zwischen verschiedenen Komponenten auszudrücken [Nor99b, S. 11]. Dies macht sich insbesondere in den Bereichen bemerkbar, in denen sich objektorientierte Ansätze durchgesetzt haben, beispielsweise bei der Programmierung graphischen Benutzeroberflächen (GUI) und verteilter Anwendungen (Corba [cor], JavaRMI [rmi00]). In Curry bieten die Kommunikation über freie Variablen, verbunden mit nebenläufiger Auswertung, und ein Portkonzept, das in Kapitel 2.2.2 vorgestellt wird, zwar zusätzliche Interaktionsmöglichkeiten, doch in der Praxis zeigt sich, dass auch hier Objekt-ähnliche Techniken von Nutzen sind (vgl. [Han99], [Han00b]).

1.3.3. Wiederverwendung von Code

Funktionale Programmiersprachen unterstützen die Wiederverwendung von Code mit einer Reihe von Mechanismen, wie parametrischem Polymorphismus, Funktionen höherer Ordnung und verzögerter Auswertung [HS95]:

- Polymorphe Funktionen wie `sort` lassen sich auf unterschiedliche Typen anwenden.
- Rekursive Muster lassen sich als Funktionen höherer Ordnung, wie beispielsweise `map`, `filter` oder `foldl` implementieren.
- Mehrere Schleifen, die bis zu einem festen Wert n laufen, lassen sich durch Funktionen ersetzen, die unendliche Listen produzieren, und `take` n übernimmt das Zählen der Iterationen.

Die Wiederverwendung von Funktionen wird jedoch stark erschwert, wenn Datentypen nachträglich um zusätzliche Komponenten erweitert werden sollen. Modifiziert man den algebraischen Datentyp, so müssen auch alle Funktionen angepasst werden, die mit diesen Typen arbeiten. Erzeugt man einen zusätzlichen Typ, so muss jede Funktion über diesem Typ neu definiert werden. Auch das Klassenkonzept aus Haskell bietet hier keine Abhilfe. Es ermöglicht lediglich das Überladen von Funktionen, d. h. die Verwendung gleicher Namen für verschiedene Funktionen.

Die zusätzlichen Möglichkeiten der Interaktion, der Wiederverwendung von Code durch Vererbung und der Modellierung von Dingen der realen Welt motivieren die Erweiterung deklarativer Programmiersprachen um objektorientierte Konzepte. Im folgenden Abschnitt betrachten wir einige solcher Erweiterungen.

1.4. Objektorientierte Erweiterungen deklarativer Programmiersprachen

Im Bereich der funktionalen Programmiersprachen gibt es ein Fülle von Ansätzen, Objektorientierung in bestehende Sprachen zu integrieren. Sprachen wie Haskell und ML sind bereits mehrfach um objektorientierte Merkmale erweitert worden. Die verschiedenen Vorgehensweisen unterscheiden sich stark in der Zielsetzung, im Umfang der realisierten Features und in der zur Realisierung verwendeten Technik. Auch logische und insbesondere nebenläufig logische Programmiersprachen wurden um Objekte, Klassen und Vererbung erweitert. Wir betrachten im Folgenden einige solcher Erweiterungen.

1.4.1. Haskell++

Haskell++ [HS95] erweitert Haskell um **Objektklassen**. Wie eine Typklasse legt eine Objektklasse eine Menge von Funktionen fest, die jede Instanz der Objektklasse implementieren muss. Im Gegensatz zu Typklassen bietet sich bei Instanzen von Objektklassen jedoch die Möglichkeit, Funktionen einer anderen Instanz derselben Typklasse zu erben. Instanzen von Haskell's Typklassen hingegen können nur in der Klasse festgelegte Funktionen übernehmen. Instanzen von Objektklassen spielen die Rolle von Klassen in konventionellen objektorientierten Programmiersprachen, Objektklassen selber sind also Metaklassen. Ein Objekt ist das Element einer Instanz einer Objektklasse, also ein gewöhnlicher Wert eines abstrakten Datentyps. Funktionen, die den Zustand eines Objekts verändern sollen, liefern ein neues Objekt zurück.

Haskell++ unterstützt eine eingeschränkte Form von Mehrfachvererbung und virtuelle Methoden. Haskell++ beinhaltet jedoch keinen Subtyppolymorphismus und kein dynamisches Binden. Es ist dadurch beispielsweise nicht möglich, eine Liste zu erzeugen, die Elemente verschiedener Instanzen einer Objektklasse beinhaltet.

Die Ansätze bei der Entwicklung von ObjectCurry und Haskell++ unterscheiden sich stark voneinander. In Haskell++ wurde das System der Typklassen von Haskell erweitert, Methoden sind als Funktionen implementiert und Objekte sind Elemente eines Datentyps, besitzen also keinen internen Zustand. Vorrangiges Ziel bei der Entwicklung von Haskell++ war es, eine minimale Erweiterung zu schaffen, die das Erben von Funktionen erlaubt. Ziel von ObjectCurry hingegen war es, echte Objekte mit Zustand im Sinne objektorientierter Programmiersprachen zu realisieren und eine mindestens so hohe Flexibilität wie in einer konventionellen objektorientierten Sprache zu erreichen.

1.4.2. O'Haskell

O'Haskell [Nor99b] ist eine Erweiterung von Haskell, die die Erstellung von nebenläufigen, robusten Anwendungen gestattet. Dazu stellt O'Haskell **reaktive Objekte** zur Verfügung. Ein reaktives Objekt ist gleichzeitig ein Objekt mit Zustand im Sinne der objektorientierten Programmierung und ein nebenläufiger Prozess: Es empfängt Nachrichten und reagiert auf diese. Ein reaktives Objekt kann einen Prozess, von dem es eine Nachricht erhalten hat, nicht suspendieren. Dadurch ist die Reaktivität des gesamten Systems zu jedem Zeitpunkt gewährleistet.

O'Haskell führt Records ein und verwendet Monaden um Objekte mit Zuständen zu implementieren. Zusätzlich enthält O'Haskell ein komplexes Typsystem mit Subtyppolymorphismus über abstrakten Datentypen und Records. Objekte sind Bürger erster Klasse. Insgesamt stehen dem Programmierer damit alle Möglichkeiten zur Verfügung, die objektorientierte und funktionale Programmierung erlauben. Darüber hinaus bietet O'Haskell eine Reihe von Synergieeffekten und nicht zuletzt die Möglichkeit, nebenläufige Anwendungen zu erstellen.

Auch Objekte in Curry besitzen einen Zustand und werden nebenläufig abgearbeitet. In der aktuellen Curry-Implementierung stehen Monaden jedoch nicht zur Verfügung. Die Implementierung stützt sich daher auf die nebenläufigen logischen Eigenschaften von Curry und verwendet Ports für das Senden von Nachrichten. Das Typsystem von ObjectCurry ist wesentlich einfacher als das von O'Haskell, da Subtypen nur über Objekten und Nachrichten erlaubt werden. Diese Einschränkung reicht für unsere Zielsetzung, die Anwendung objektorientierter Techniken so weit wie möglich zu ermöglichen, jedoch völlig aus. Durch die Verwendung von Ports eignet sich ObjectCurry im Gegensatz zu O'Haskell auch zur Erstellung verteilter Anwendungen. Objekte in ObjectCurry sind nicht reaktiv, da sie blockiert werden und selbst sendende Prozesse blocken können.

Wir übernehmen aus O'Haskell den Begriff Template zur Bezeichnung von Klassen, um

Verwechslungen mit Haskell's Typklassen zu vermeiden.

1.4.3. O'Caml

Eine konservative Erweiterung der Sprache ML ist O'Caml [Ler96]. O'Caml unterstützt die meisten objektorientierten Ansätze. Objekte lassen sich in O'Caml direkt oder als Instanzen von Klassen erzeugen. Realisiert werden Objekte durch Records, die die Instanzvariablen und Methoden enthalten. Für die Transformation des Zustandes eines Objekts werden die imperativen Elemente von O'Caml ausgenutzt. O'Caml unterstützt sowohl Einfach- als auch Mehrfachvererbung. Eine geerbte Methode kann sowohl auf Objekte der Ober- als auch auf Objekte der Unterklasse angewendet werden. O'Caml verwendet das Typsystem von ML. Es ist daher nicht möglich, Objekte unterschiedlicher Klassen in einer Liste zu mischen, da sie unterschiedliche Typen besitzen. Diese Art von Subtyppolymorphismus kann aber explizit durch die Verwendung von **Coercions** erreicht werden: Wenn ein Objekt vom Typ t_2 als ein Objekt vom Typ t_1 angesehen werden soll, dann erreicht man dies in O'Caml durch die Angabe der Coercion t_2 `:>` $t1$ hinter dem Objekt. Die Verwendung einer solchen Coercion ist nur sicher, wenn t_2 von t_1 erbt. Anderenfalls können Laufzeitfehler auftreten.

In ObjectCurry möchten wir auf Coercions verzichten, da diese in konventionellen objektorientierten Programmiersprachen nicht vorkommen, und wir es in Curry wie in Haskell gewöhnt sind, auf die Angabe von Typen in den meisten Fällen verzichten zu können.

1.4.4. Vulcan

Shapiro und Takeuchi [ST83] beschreiben, wie sich Objekte in Concurrent Prolog modellieren lassen. Sie realisieren dabei ein Objekt durch einen Prozess, der sich selbst rekursiv aufruft, und den Zustand des Objekts in seinen Argumenten hält. Objekte kommunizieren untereinander über gemeinsame Variablen (engl. *shared variables*): Ein Prozess wartet solange, bis eine nicht instantiierte gemeinsame Variable durch einen anderen Prozess, den Sender, gebunden wird. Dieser bindet die Variable an ein Paar. Die erste Komponente dieses Paares ist die Nachricht, die er an das Objekt senden möchte. Die zweite Komponente ist eine neue, nicht instantiierte Variable, über die weitere Nachrichten an das Objekt gesendet werden können.

Um eine **Antwort** an den Sender einer Nachricht zurückzugeben, verwenden Shapiro und Takeuchi **unvollständige Nachrichten**. Eine unvollständige Nachricht enthält eine nicht instantiierte, logische Variable. Das Objekt, das eine unvollständige Nachricht erhält, bindet diese Variable an die Antwort, die es zurückgeben möchte.

Die Sprache Vulcan [KTMB87] nutzt Shapiros Technik, indem sie Concurrent Prolog um spezielle Syntax zur Definition von Klassen erweitert. Ein Objekt kann in Vulcan als Instanz einer solchen Klasse erzeugt werden. Vulcan unterstützt Vererbung, ist aber ungetypt, da auch Concurrent Prolog eine ungetypte Sprache ist.

In ObjectCurry übernehmen wir die Ideen, Objekte in Form von Constraints zu realisieren, sowie unvollständige Nachrichten für die Rückgabe von Werten zu verwenden. Anstelle von logischen Variablen verwenden wir jedoch Ports für die Kommunikation zwischen und mit Objekten.

2. Objektbasierte Programmierung in Curry

Nachdem die Grundzüge der Sprache Curry vorgestellt wurden, ist es das Ziel dieses Kapitels, eine Technik vorzustellen, die aufzeigt, wie bereits in reinem Curry ohne zusätzliche Erweiterungen objektbasiert programmiert werden kann. Ein Objekt wird dabei als ein Prädikat aufgefasst, das einen Strom von Nachrichten verarbeitet. Diese Technik geht ursprünglich auf E. Shapiro und A. Takeuchi [ST83] zurück und findet beispielsweise in Vulcan (s. Kapitel 1.4.4, S. 13) Anwendung. Im Rahmen von Curry wurde sie bereits in [Han99] und [Han00b] vorgestellt.

Auf den Aspekt der **Vererbung** werden wir in diesem Kapitel nicht eingehen, sondern ihn bis zur Vorstellung der Sprache ObjectCurry in Kapitel 3 zurückstellen.

2.1. Objekte in Curry

Wir möchten in Curry **Objekte** modellieren, die sich wie Objekte in konventionellen objektorientierten Programmiersprachen verhalten. Insbesondere sollen sie die folgenden Eigenschaften aufweisen:

1. Ein Objekt kann einen **Zustand** besitzen.
2. Ein Objekt reagiert auf **Nachrichten**, die an dieses gesendet werden.
3. Der Zustand des Objekts ist **gekapselt**, d. h. er kann nur durch das Senden von Nachrichten verändert werden.

Die **Reaktion** eines Objekts auf eingehende Nachrichten kann z. B. im Wechsel seines Zustands, aber auch im Versenden weiterer Nachrichten an andere Objekte bestehen.

Aus dem Bereich der nebenläufigen logischen Programmiersprachen ist eine Technik bekannt, mit deren Hilfe Objekte mit Zustand modelliert werden können. Diese Technik kann leicht auf Curry übertragen werden. Dabei wird ein Objekt als ein Constraint bzw. Prädikat aufgefasst, das einen **Strom** von eingehenden Nachrichten verarbeitet. Als Eingabe erhält die Funktion den aktuellen Zustand des Objekts sowie einen Strom von Nachrichten. Der Nachrichtenstrom wird durch eine Liste dargestellt. Die Funktion liest das erste Element dieser Liste und ruft sich dann rekursiv mit einem neuen Zustand und den noch nicht verarbeiteten Nachrichten auf. Der allgemeine Typ eines Objekts lautet wie folgt:

$$o :: State \rightarrow [messageType] \rightarrow \texttt{Success} \tag{2.1}$$

State steht für den Typ des internen Zustands des Objekts und *messageType* für den Typ der Nachrichten, die an dieses Objekt gesendet werden können. Es bietet sich an, einen speziellen abstrakten Datentyp zu diesem Zwecke zu definieren, welcher für jede Art von Nachrichten einen eigenen Datenkonstruktor enthält. Wir nennen diesen Datentyp im weiteren Verlauf der Arbeit **Nachrichtentyp**.

2.1.1. Antworten

In den meisten konventionellen objektorientierten Programmiersprachen wie **Eiffel**, **Smalltalk**, **C++** oder **Java** geben Objekte nach Erhalt einer Nachricht und einer entsprechenden Reaktion häufig eine **Antwort**, die der Sender der Nachricht in Form eines Funktionsergebnisses zurückerhält. Ein solcher Mechanismus lässt sich einerseits realisieren, indem der Sender eine zusätzliche Information über sich selbst mitschickt, so dass das empfangende Objekt eine Antwort als Nachricht zu einem späteren Zeitpunkt an den Sender zurücksenden kann. In Curry ist es jedoch auch möglich, Anworten wie Shapiro und Takeuchi durch die Verwendung **unvollständiger Nachrichten** (s. S. 13) zurückzuliefern: Als Bestandteil der Nachricht wird eine freie logische Variable mitgeschickt, die der Empfänger an die Antwort bindet. Dem ursprünglichen Sender steht diese Antwort dann in Form der logischen Variable zur Verfügung.

2.1.2. Beispiel: Ein Zähler

Als Beispiel betrachten wir einen einfachen **Zähler** (vgl. [Han99]). Dieser wird als ein Objekt realisiert, das einen Integer-Wert als internen Zustand besitzt und auf die Nachrichten `Inc` zum Inkrementieren, `Set s` zum Setzen und `Get v` zum Abfragen des Zustandes reagiert. Als Bestandteil der Nachricht `Set s` wird ein Integer-Wert `s` mitgeschickt. Bei der Nachricht `Get v` wird von der oben beschriebenen Möglichkeit Gebrauch gemacht, Antworten durch das Mitschicken von freien Variablen zurückzugeben. `Get` wird daher immer auf eine freie logische Variable angewandt werden. Wenn wir den Nachrichtentyp als einen abstrakten Datentyp `counterMessage` definieren, lässt sich die Funktion `counter`, die den Zähler wie oben beschrieben realisiert, bequem mit Hilfe des **Pattern-Matching** angeben:

Bsp. 2.1.1 (Zähler in Curry)

```
data CounterMessage = Inc | Set Int | Get Int

counter :: Int -> [CounterMessage] -> Success
counter eval rigid
counter x (Inc   : ms) = counter (x+1) ms
counter _ (Set s : ms) = counter s ms
counter x (Get v : ms) = v =:= x & counter x ms
counter _ [] = success
```

Der angegebene Typ von `counter` entspricht, wie sich leicht überprüfen lässt, dem oben vorgestellten allgemeinen Typ eines Objekts.

`(Inc : ms)` ist die übliche Notation zur Darstellung von Listen, wobei `Inc` den Kopf und `ms` den Rest der Liste bezeichnet. `_` ist die **anonyme Variable**, die auf jeden Wert passt. Die linken Seiten der vier Gleichungen sind nicht überlappend, daher ist die Funktion `counter` deterministisch. Dies ist insofern wichtig, da ein nicht-deterministisches Objekt keinen Sinn macht.

Deklarativ lässt sich die erste Regel wie folgt interpretieren: Ein Zähler im Zustand `x`, der die Nachricht `Inc` sowie eine Liste mit weiteren Nachrichten erhält, verhält sich wie ein Zähler im Zustand `(x+1)`, der die weiteren Nachrichten empfängt. Aus objektorientierter Sicht wird der Zustand des Zählers beim Empfangen der Nachricht `Inc` erhöht. Analog verhält es sich mit der Nachricht `Set s`: Ein Zähler im Zustand `x`, der die Nachricht `Set s` und eine Liste mit weiteren Nachrichten erhält, verhält sich wie ein Zähler im Zustand `s`, der die übrigen Nachrichten empfängt. Die Nachricht `Set s` bewirkt ein Setzen des Zustands auf `s`.

Die Reaktion auf die Nachricht `Get v` besteht darin, dass die mitgeschickte freie Variable `v` an den internen Zustand des Zählers gebunden wird. Realisiert wird dies mit Hilfe des Gleichheitsconstraints `=:=`, der die Ausdrücke seiner linken und rechten Seite soweit wie möglich auswertet und miteinander unifiziert. An dieser Stelle sei darauf hingewiesen, dass in diesem Rahmen nicht sichergestellt werden kann, dass es sich beim Erhalt der Nachricht `Get v` bei der Variablen `v` tatsächlich um eine freie Variable handelt. Sollte dies nicht der Fall sein und die Unifikation mit dem Zustand des Zählers fehlschlagen, so ist das gesamte Prädikat, das das Zählerobjekt repräsentiert, nicht erfüllbar, denn es gibt keine andere Möglichkeit, auf die Nachricht `Get v` zu reagieren. Das Objekt würde in diesem Fall mit einem Fehlschlag beendet.

Die Angabe der Auswertungsstrategie `rigid` bewirkt, dass ein Ausdruck der Form `counter` $n\ \ s$ nur dann reduziert werden kann, wenn insbesondere der Nachrichtenstrom s gebunden ist (vgl. Kapitel 1.2.3).

Die Auswertung des Ausdrucks `counter 0 s` erzeugt einen Zähler mit dem Zustand `0`. Nachrichten lassen sich durch Binden der Variable `s` an die Liste der zu sendenden Nachrichten verschicken:

Das Prädikat

```
let s free in (counter 40 s) & s =:= [Inc,Inc,Get x,Inc]
```

wird zu `success` ausgewertet, und `x` an den Wert `42` gebunden.

Das Binden von `s` kann auch inkrementell geschehen, indem `s` mit einer Liste unifiziert wird, von der nur das erste Element gebunden ist und der Rest aus einer neuen, freien Variable besteht:

```
let s1,s2,s3 free in
  counter 41 s1 & s1 =:= (Inc:s2) & s2 =:= (Get x:s3) & s3 =:= []
```

Es ist unbedingt erforderlich, den Nachrichtenstrom zu schließen, da das Objekt sonst auf weitere Nachrichten wartet und nicht zu `success` reduziert werden kann. Objekte mit unendlicher Lebensdauer machen in verteilten Systemen einen Sinn; diesem Thema widmet sich ausführlich Kapitel 4.

2.1.3. Datenkapselung

Bei Betrachtung der bisherigen Vorgehensweise wird klar, dass der Zugriff auf ein einmal erzeugtes Objekt immer nur indirekt erfolgt, nämlich indem Nachrichten an dieses geschickt werden. Auf diese Weise ist sichergestellt, dass der interne Zustand eines Objekts vollständig von der Außenwelt abgekapselt ist.

Die auf diese Weise erzeugten Objekte erfüllen bereits die zu Beginn von Kapitel 2.1 geforderten Eigenschaften: Ein Objekt reagiert auf Nachrichten und besitzt einen gekapselten Zustand. Jedoch ist es erforderlich, den Mechanismus des Versendens von Nachrichten näher zu betrachten, wie in den folgenden Abschnitten deutlich wird.

2.2. Kommunikationsmedien

Das Senden einer Nachricht erfolgt über ein **Kommunikationsmedium**:

Def. 2.2.1 (Kommunikationsmedium)
*Ein **Kommunikationsmedium** ist ein Datentyp, der Nachrichten zwischen als Objekten agierenden Prozessen überträgt (nach [JMH93]).*

Das Kommunikationsmedium dient als Schnittstelle zu einem Objekt. Für den Programmierer stellt das Kommunikationsmedium das eigentliche **Objekt** dar, denn an dieses sendet er Nachrichten. Bisher diente uns immer ein Strom, realisiert durch eine Liste, als Kommunikationsmedium.

Janson et al. stellen in [JMH93] eine Reihe von Forderungen an ein solches Kommunikationsmedium, von denen die folgenden für uns von Bedeutung sind:

- Die Anzahl der Operationen, die intern nötig sind, um eine Nachricht zu versenden, sollte konstant sein, d. h. unabhängig von der Anzahl der Sender. Diese Eigenschaft wird als **constant delay property** bezeichnet.

- Beim Verschicken von mehreren Nachrichten an ein Objekt sollte dieses durch die Benutzung eines einzigen Kommunikationsmediums möglich sein. In konventionellen objektorientierten Programmiersprachen scheint diese Bedingung trivial, jedoch nicht in Sprachen wie Curry, in denen freie Variablen nur einmal gebunden werden können.

Des Weiteren fordern Janson et al., dass Speicherplatz, den Nachrichten und Objekte blockieren, automatisch freigegeben werden kann, wenn diese nicht mehr benötigt werden.

Im Folgenden betrachten wir zunächst Ströme als Kommunikationsmedium und werden sehen, dass diese die oben geforderten Eigenschaften nicht erfüllen.

2.2.1. Ströme als Kommunikationsmedium

Bisher haben wir Ströme als Kommunikationsmedium eingesetzt. Das Schicken von Nachrichten geschah durch inkrementelles Binden der Stromvariablen durch den Sender. Dieser

Mechanismus funktioniert jedoch nur, solange nur ein einziger Sender Zugriff auf den Strom hat: Nehmen wir an, zwei Sender würden sich die Stromvariable teilen. Verschickt nun einer der beiden Sender eine Nachricht, indem er die Stromvariable bindet, dann ist diese nicht mehr frei, und der zweite Sender kann sie nicht mehr zum Versenden von Nachrichten verwenden. Er kennt auch nicht die vom ersten Sender als Fortsetzung des Nachrichtenstroms erzeugte neue, freie Variable.

Wenn nicht das Objekt selber mit mehreren Schnittstellen, d. h. mehreren Strömen nach außen verbunden werden soll, wird ein **Merger** benötigt, der mehrere Ströme zu einem Strom verschmelzt. Ein fairer Merger kann in Curry wie folgt implementiert werden ([Han99]):

```
merge eval choice
merge (x:xs) ys       = x : merge xs ys
merge xs     (y:ys)   = y : merge xs ys
merge []     ys       = ys
merge xs     []       = xs
```

Die Annotation `choice` bewirkt, dass maximal eine Regel beim Aufruf von `merge` angewendet wird, auch wenn weitere Regeln anwendbar sind.

Mit Hilfe der `merge`-Funktion lässt sich ein Counter-Objekt erzeugen, das Nachrichten von verschiedenen Quellen akzeptiert:

```
let s1,s2 free in counter 0 (merge s1 s2) & client1 s1 & client2 s2
```

Die zweite der auf Seite 18 an Kommunikationsmedien gestellten Eigenschaften ist erfüllt: Ein Sender schickt Nachrichten immer über denselben, ihm exklusiv zur Verfügung stehenden Strom. Auch das Freigeben des von Nachrichten und Objekten belegten Speicherplatzes macht keine Probleme, denn es werden nur Konstrukte aus dem Sprachumfang von Curry verwendet, so dass der Garbagecollector die Speicherverwaltung übernimmt.

Um n verschiedene Sender mit einem Objekt zu verbinden, werden jedoch mindestens $\log n$ Merger benötigt. Damit ist die constant delay property verletzt.

Der Einsatz von `merge` birgt zudem einen weiteren Nachteil, denn die Verwendung des nicht-deterministischen Choice zerstört die Vollständigkeitseigenschaften der operationalen Semantik und die gewohnte deklarative Lesart ([Han97],[Han99]).

2.2.2. Ports als Kommunikationsmedium

Janson et al. [JMH93] schlagen den Einsatz von Ports als Kommunikationsmedium vor, um die angesprochenen Probleme zu vermeiden.

Def. 2.2.2 *Ein **Port** ist ein Constraint zwischen einer Multimenge und einer Liste, der erfüllt ist, wenn alle Elemente der Multimenge in der Liste vorkommen und umgekehrt.*

Ein Port kann mit Hilfe des Constraints `openPort p s` erzeugt werden. Dabei sind `p` und `s` freie Variablen. Bei der Auswertung des Constraints wird `p` an eine Multimenge und `s` an eine Liste gebunden, und beide durch einen Port miteinander verknüpft. Das Senden von Nachrichten erfolgt durch das Einfügen von Elementen in die Multimenge `p`. Der Port bewirkt, dass diese auch in der Liste `s` enthalten sind. Aus diesem Grund identifizieren wir `p` mit dem Port selbst und `s` mit der Liste der eintreffenden Nachrichten. Der Typ eines Ports, über den Nachrichten des Typs `a` versendet werden, wird mit `Port a` notiert. `openPort` besitzt den Typ

```
openPort :: Port a -> [a] -> Success
```

Das Senden von Nachrichten erfolgt über einen weiteren Constraint: `send m p` sendet die Nachricht `m` an den Port `p`. Semantisch bedeutet dies, dass die Nachricht `m` in die Multimenge und die mit ihr über den Port verbundene Liste eingefügt wird. Das Senden geschieht strikt, d. h., dass die Nachricht `m` vor dem Versenden zu einem Grundterm reduziert wird. Mit Hilfe von Ports lässt sich das Beispiel des Zählers mit zwei Sendern wie folgt aufschreiben:

```
openPort p s &> counter 0 s & client1 p & client2 p
```

Die Sender bedienen sich zum Senden der Nachricht des Ports `p` sowie des `send`-Constraints und haben keinen direkten Zugriff auf den Nachrichtenstrom:

```
client1 p = send Inc p
```

Ein Objekt wurde bislang beendet, indem der Nachrichtenstrom geschlossen wurde. Dies ist einem Sender nun nicht mehr möglich, da er nur Zugriff auf den Port hat. Daher wählen wir eine andere Möglichkeit, ein Objekt zu beenden: Wir führen eine spezielle Nachricht `Stop` ein. Ein Objekt beendet sich, wenn es diese Nachricht empfängt:

```
counter _ (Stop:_) = success
```

Die Verwendung von Ports als Kommunikationsmedium löst die Probleme, die durch das Vorhandensein mehrerer Sender entstehen. Uns steht damit eine Technik zur Verfügung, die objektbasierte Programmierung in Curry ermöglicht.

Da Nachrichten als Elemente eines abstrakten Datentyps auch zur Laufzeit eines Programms existieren, ist es mit der vorgestellten Technik einfach, Objekte auf mehrere Rechner zu verteilen. Im Detail werden wir diese Möglichkeit in Kapitel 4 behandeln. In vielen konventionellen objektorientierten Programmiersprachen wie C++, Java oder Eiffel, aber auch in Sprachen wie Haskell++ oder O'Haskell, besteht das Senden von Nachrichten aus dem Aufruf einer Funktion oder Prozedur. Nachrichten existieren nur auf einer rein abstrakten Ebene. Dadurch gestaltet sich die Verteilung von Objekten hier weitaus komplizierter und führt zu Techniken wie JavaRMI [rmi00] oder Corba [cor].

Nicht eingegangen sind wir bisher auf das objektorientierte Konzept der Vererbung. Auch gestattet das Typsystem von Curry in einigen Anwendungen nicht die gleiche Flexibilität wie der Subtyppolymorphismus konventioneller objektorientierter Programmiersprachen. Beide Fragen werden in den Kapiteln 3 und 5 aufgegriffen und vertieft.

3. ObjectCurry

In Kapitel 2 wurde eine Technik vorgestellt, die es ermöglicht, mit Hilfe von Ports in Curry Objekte zu erzeugen, die einen Zustand besitzen und auf Nachrichten reagieren, die von beliebig vielen Sendern stammen können. Die Anwendung dieser Programmiertechnik birgt jedoch auch Nachteile:

- Die Übergabe des modifizierten Zustands eines Objekts beim rekursiven Aufruf ist für den Programmierer sehr mühsam.
- Rein technisch notwendiger Code erschwert die Lesbarkeit des Programms.
- Jede zusätzliche Zeile Code birgt das Risiko von Tippfehlern.

Aus diesem Grunde bietet es sich an, die Syntax von Curry zu erweitern, und die bisherige Programmiertechnik intern zur Implementierung zu verwenden.

In diesem Kapitel wird die Sprache **ObjectCurry** vorgestellt, bei der es sich um eine Erweiterung der Sprache Curry handelt. Die Erweiterung ist konservativ, d. h. jedes Curry-Programm ist gleichzeitig auch ein gültiges ObjectCurry-Programm und besitzt in beiden Sprachen dieselbe Semantik. Ziel bei der Entwicklung von ObjectCurry war es, zusätzliche Syntax zur Verfügung zu stellen, die es erlaubt, möglichst viele Möglichkeiten konventioneller objektorientierter Programmiersprachen einfach anwendbar zu machen. Dazu zählen insbesondere:

- Definition von Klassen
- Erzeugung von Objekten
- Vererbung von Klassendefinitionen
- verteilte Objekte (Kap. 4)
- Mechanismen zur Synchronisation (Kap. 4)
- Subtyppolymorphismus über Objekten und Nachrichten (Kap. 5)

3.1. Die Sprache ObjectCurry

Wir beschreiben die Syntax von ObjectCurry zunächst informell an Hand von Beispielen. Eine Grammatik, die die Syntax vollständig festlegt, findet sich auf Seite 37.

Jedes Curry-Programm ist im Wesentlichen eine Menge von Funktionsdefinitionen. Ein ObjectCurry-Programm ist eine Menge von Funktions- und Templatedefinitionen. Ein

Template ist eine Schablone für Objekte. Es beschreibt insbesondere deren Zustand und wie dieses auf Nachrichten reagiert. In konventionellen objektorientierten Programmiersprachen werden diese Schablonen üblicherweise Klassen genannt. Da die Syntax von Curry sehr stark an die von Haskell angelehnt ist, und um Verwechslungen mit den aus Haskell bekannten Typklassen zu vermeiden, verwenden wir in Anlehnung an O'Haskell (s. S. 12) den Begriff „Template".

3.1.1. Definition von Templates

In Kapitel 2 wurde ein einfaches Zählerobjekt vorgestellt. In ObjectCurry sieht ein Template zur Erzeugung eines solchen Objekts wie folgt aus:

Bsp. 3.1.1 (Zähler in ObjectCurry)

```
template Counter =              -- Kopf
constructor                     -- Initialisierungsteil
  counter init = x := init
methods                         -- Rumpf
  Inc   = x := x + 1
  Set v = x := v
  Get v = v =:= x
```

Eine Templatedefinition besteht aus drei Teilen:

1. Der **Kopf** eines Templates enthält den Namen des Templates (`Counter`). Dieser ist gleichzeitig der Typ des Templates. Ähnlich wie bei einer data-Deklaration spielt er nur innerhalb von Typausdrücken eine Rolle.

2. Das reservierte Schlüsselwort `constructor` leitet den **Initialisierungsteil** ein. In diesem wird die **Konstruktorfunktion** des Templates definiert. Diese ermöglicht die Erzeugung von Instanzen des Templates und übernimmt damit die Rolle des Konstruktors aus konventionellen objektorientierten Programmiersprachen. Die Konstruktorfunktion muss aus genau einer definierenden Gleichung bestehen. Die linke Seite der Konstruktorfunktion entspricht der linken Seite einer üblichen Curry-Funktionsdefinition. Die rechte Seite besteht aus einer Reihe von **Zuweisungen** der Form x `:=` e. x ist hierbei der Name einer Variable und e ein beliebiger Ausdruck. Die Anwendung der **Offside-Rule** ermöglicht es, dass mehrere Zuweisungen untereinander notiert werden können, und nicht durch Kommata oder andere Trennzeichen voneinander getrennt werden müssen. Durch die Zuweisungen werden die Instanzvariablen des Templates und deren Initialwerte festgelegt. Instanzen des oben definierten Templates besitzen die Instanzvariable `x` mit dem Startwert `init`.

3. Im **Rumpf** wird beschrieben, auf welche Weise das Objekt auf Nachrichten reagiert. Zu diesem Zwecke werden so genannte **Methoden** definiert. Eine Methodendefinition hat die Form $message\, p_1 \ldots p_n$ `=` r. Der Rumpf eines Templates kann beliebig viele Methodendefinitionen beinhalten. Diese werden mit Hilfe der Offside-Rule untereinander gesetzt. Eine Methode deklariert zum einen die n-stellige Nachricht *message*, zum anderen legt sie die Reaktion des Objekts auf diese Nachricht fest. *message* ist der Name der Nachricht. Da Nachrichten, wie in Kapitel 2 beschrieben, Elemente eines abstrakten Datentyps sind, werden diese nach der Haskell-Konvention

im allgemeinen groß geschrieben. $p_1 \dots p_n$ können wie bei einer Funktionsdefinition Variablen oder Muster sein.

Die rechte Seite r einer Methodendefinition besteht aus einer Menge von Zuweisungen und Constraints, welche unter Beachtung der Offside-Rule untereinander notiert werden. Sie beschreiben die Reaktion des Objekts auf den Erhalt der Nachricht *message*.

Ein Objekt kann auf verschiedene Weisen auf eine Nachricht reagieren. Eine Möglichkeit besteht darin, dass es in einen anderen Zustand übergeht. Eine Zustandstransformation lässt sich durch eine Menge von Zuweisungen beschreiben:

Def. 3.1.2 (Zustandstransformation)
Sei A eine Menge von Zuweisungen der Form $v_i = e_i$. Die Menge A beschreibt die ***Zustandstransformation*** $(v_1, \dots, v_n) \mapsto (v'_1, \dots, v'_n)$, *falls*

$$v'_i = \begin{cases} e_i & \textit{falls } v_i := e_i \in A \\ v_i & \textit{sonst} \end{cases}$$

Die Reihenfolge, in der Zuweisungen auf der rechten Seite einer Methodendefinition auftreten, spielt keine Rolle.

Bsp. 3.1.3 *Die Zustandstransformationen*

```
x := 5
y := x
```

und

```
y := x
x := 5
```

sind äquivalent: Anders als in imperativen Programmiersprachen überführen beide den Zustand (a, b) in den Zustand $(5, a)$, $a, b \in \mathcal{N}$[1].

Weitere Reaktionen können durch die Angabe von Constraints beschrieben werden. Constraints sind Ausdrücke des Typs `Success`. Der Gleichheitsconstraint `=:=` kann beispielsweise dazu verwendet werden, Rückgabevariablen mit einem Wert zu belegen, so z. B. in der Methode `Get` in Beispiel 3.1.1. Bevor die durch die Zuweisungen beschriebene Zustandstransformation ausgeführt wird, müssen alle auf der rechten Seite auftretenden Constraints gelöst, d. h. zu `success` ausgewertet werden. Wir werden später sehen, was passiert, wenn dies nicht möglich ist.

Eine Templatedefinition deklariert auf globaler Ebene die Konstruktorfunktion und die Nachrichten, die durch die Methoden bestimmt werden. Eine n-stellige Konstruktorfunktion c eines Templates T besitzt den Typ

$$\tau_1 \texttt{ -> } \dots \texttt{ -> } \tau_n \texttt{ -> } \texttt{Constructor}\ T$$

[1] $\mathcal{N}$ sei die Menge der natürlichen Zahlen.

Eine n-stellige Nachricht m eines Templates T besitzt den Typ

$$\tau_1 \texttt{ -> } \ldots \texttt{ -> } \tau_n \texttt{ -> } \texttt{Message}\, T$$

Dabei sind τ_i beliebige Typausdrücke und `Constructor` sowie `Message` vordefinierte, einstellige Datentypkonstruktoren. In Kapitel 5 werden wir die Typen von Nachrichten verändern, um Subtyppolymorphismus zu ermöglichen.

3.1.2. Objekterzeugung und Senden von Nachrichten

Objekte sind Instanzen von Templates. Die Erzeugung von Objekten, d. h. die Bildung von neuen Instanzen eines Templates, erfolgt mit Hilfe einer vordefinierten Funktion `new`. `new (counter 41) x` erzeugt ein Counterobjekt mit dem Anfangszustand 42 und bindet `x` an einen neuen Port, über den Nachrichten an den Zähler gesendet werden können. Aus Sicht des Programmierers bezeichnet `x` das erzeugte Objekt. `new` besitzt den Typ

```
Constructor a -> Object a -> Success
```

Zum Senden von Nachrichten wird die Funktion `sendMsg` verwendet. Soll beispielsweise die Nachricht `Inc` an das oben erzeugte Zählerobjekt `x` gesendet werden, so geschieht dies mit `sendMsg Inc x`. In einem ersten Ansatz geben wir der Funktion `sendMsg` den Typ

```
Message a -> Object a -> Success
```

Auch die Typen von `new` und `sendMsg` werden wir in Kapitel 5 modifizieren.

Falls das Objekt als Reaktion auf eine Nachricht eine Antwort zurückliefern soll, so wird dies über das Mitschicken einer freien Variable realisiert. Diese wird durch einen Constraint in der entsprechenden Methode gebunden, wie wir es bereits auf Seite 16 gesehen haben.

Beispiel:

```
test | new (counter 41) cobj
       & (sendMsg Inc cobj &> sendMsg (Get r) cobj &> sendMsg Stop cobj)
     = r
  where r,cobj free
```

Der Versuch, den Ausdruck `test` zu reduzieren, verlangt zunächst, dass der angegebene Constraint gelöst wird. `new (counter 41) cobj` erzeugt ein neues Objekt, das die Nachrichten aus einem zunächst leeren Stream verarbeitet, und einen Port, der mit diesem Stream verbunden ist. Die Variable `cobj` wird mit diesem Port verbunden. Aus Sicht des Programmierers repräsentiert `cobj` das erzeugte Zählerobjekt. Durch Senden von Nachrichten an `cobj` wird der Nachrichtenstrom gefüllt. Die Verwendung von `&>` an Stelle von `&` bewirkt, dass die Nachrichten in der gewünschte Reihenfolge gesendet werden.

`new (counter 41) cobj` wird zu `success` ausgewertet, wenn das Objekt durch Empfang der Nachricht `Stop` beendet wird. Schlägt ein Constraint innerhalb einer Methode des Objekts fehl, so schlägt auch der `new`-Constraint fehl.

Der Constraint zum Erzeugen des Objekts und die Folge der Constraints zum Senden der Nachrichten müssen unbedingt nebenläufig ausgewertet werden, denn der Constraint `new (counter 41) cobj` kann erst zu `success` ausgewertet werden, *nachdem* die Nachricht `Stop` zum Beenden des Objekts gesendet wurde.

ObjectCurry gestattet auch die explizite Angabe von Typannotationen für die Methoden und die Konstruktorfunktion eines Templates. Die Syntax entspricht dabei der von herkömmlichen Funktionsdefinitionen. Der Rückgabewert der Konstruktorfunktion eines Templates T trägt immer den Typ `Constructor` T, der Rückgabewert jeder Methode den Typ `Message` T.

Bsp. 3.1.4 (Ein Zähler mit Typannotationen)

```
template Counter =                        -- Kopf
constructor                               -- Initialisierungsteil
  counter :: Int -> Constructor Counter
  counter init = x := init
methods                                   -- Rumpf
  Inc :: Message Counter
  Inc   = x := x + 1
  Set :: Int -> Message Counter
  Set v = x := v
  Get :: Int -> Message Counter
  Get v = v =:= x
```

Theoretisch können die Datenkonstruktoren `Message`, `Constructor` und `Object` auch in Typannotationen von Funktionsdefinitionen verwendet werden. Der ObjectCurry-Compiler muss diese dann in die Typen der internen Repräsentation umwandeln (s. Tabelle 6.1, S. 89). Dieser Mechanismus ist im Rahmen dieser Diplomarbeit jedoch nicht implementiert worden.

3.1.3. Übersetzung von Templates

Das Template zur Erzeugung von Counter-Objekten lässt sich relativ einfach in reinen Curry-Code übersetzen. Unter Zuhilfenahme von Beispielen wird das Verfahren in diesem Kapitel schrittweise verdeutlicht. Der gleiche Übersetzungsprozess wird auch von dem in Kapitel 6 vorgestellten Compiler verwendet. Wir verzichten auf die Angabe formaler Transformationsregeln, da diese sehr umfangreich sind und im Wesentlichen der Implementierung des Codegenerators entsprechen würden, der Bestandteil dieser Arbeit ist (s. Kapitel 6).

Erzeugung des Nachrichtentyps

Zunächst muss der Nachrichtentyp definiert werden. Dieser enthält die Namen aller Nachrichten, die das Counter-Objekt verstehen soll, als Datenkonstruktoren. Bei den Nachrichten `Set v` und `Get v` wird ein zusätzlicher Parameter mitgeschickt. Da unser ObjectCurry-Programm keine Typannotationen enthalten muss, ist der Typ dieser Parameter möglicherweise unbekannt. Dieser kann jedoch mittels Typinferenz vom Typchecker des ObjectCurry-

Compilers inferiert werden.

Zusätzlich zu den im Template definierten Nachrichten sollte ein Counter-Objekt auch die Nachricht `Stop` verstehen, mit der das Objekt in einen Endzustand überführt werden kann. Die Definition des Nachrichtendatentyps `Msg` ergibt sich daher wie folgt:

```
data Msg = Stop | Inc  | Get Int | Set Int
```

Die Typen der Parameter der Nachrichten `Get` und `Set` ergeben sich entweder aus den Typannotationen der Methoden, oder sie müssen durch Typinferenz ermittelt werden (s. Kapitel 5 und 6.3).

Übersetzung der Methoden

Das Verhalten eines Zähler-Objekts ist durch die im Template `Counter` festgelegten Methoden bestimmt: Zum einen besteht es in einem eventuellen Zustandswechsel, zum anderen in dem Lösen zusätzlicher Constraints. Die Übersetzung der Methoden führt uns zu einer Funktion, die als Eingabe den Zustand des Objekts und die zu verarbeitende Nachricht entgegennimmt, und den Folgezustand unter Berücksichtigung der zu lösenden Constraints zurückliefert. Im Falle des Zählers sieht diese Funktion mit dem Namen `counterAction` wie folgt aus:

```
counterAction (x) (MSG Stop)
  = Final
counterAction (x) (MSG Inc)
  = State (x + 1)
counterAction (x) (MSG (Get v))
  | v =:= x
  = State (x)
counterAction (x) (MSG (Set v))
  = State (v)
```

Wir verwenden den zusätzlichen Konstruktor `MSG` vor jeder Nachricht, um diese von synchronisierten Nachrichten, die in Kapitel 4 eingeführt werden, zu unterscheiden.

Um den Endzustand eines Objekts von solchen Zuständen, die einen Folgezustand besitzen, zu unterscheiden, wird von dem auf folgende Weise definierten Datentyp `State` Gebrauch gemacht:

```
data State a = Final | State a
```

Zum Abarbeiten eines Stroms von Nachrichten dient die Funktion `loop`. Diese berechnet mit der Action-Funktion den Nachfolgezustand des Objekts und ruft sich dann selbst rekursiv auf. Die Schleife endet, sobald der Endzustand des Objekts erreicht ist:

```
loop eval rigid
loop state action (m:ms) =
  continuation nextState ms
  where
    nextState = action state m
    continuation (State ns) ms = loop ns action ms
    continuation Final _ = success
```

Die Funktion `loop` kann nicht mit Hilfe der Funktionen `foldl` oder `foldr` aus der Curry-Prelude definiert werden, da als Abbruchkriterium das Erreichen des Endzustands dient. Würde stattdessen das Eintreffen einer bestimmten Nachricht wie beispielsweise `Stop` verwendet, wäre es im Rahmen der Vererbung nicht möglich, die Methode `Stop` zu überschreiben.

Der Startzustand eines neu erzeugten Zähler-Objekts wird über die **Initialisierungsfunktion** `counterInitState` definiert.

```
counterInitState (init) = (init)
```

Zur Erzeugung eines Objekts dient die im Template definierte Konstruktorfunktion. Diese wird als Argument an die `new`-Funktion übergeben und startet die Schleife zur Verarbeitung der Nachrichten mit dem im Template festgelegten Startzustand und den über die Action-Funktion festgelegten Reaktionen:

```
counter eval rigid
counter init = loop (counterInitState (init)) counterAction
```

Die eigentliche Objekterzeugung, d. h. das Öffnen eines neuen Ports, übernimmt die `new`-Funktion:

```
new constructor port =
  let stream free in
    openPort port stream &> constructor stream
```

Die Aufspaltung des Curry-Codes in die fünf verschiedenen Funktionen `new`, `loop`, *template*`Action`, *template*`InitState`[2] und *constructorFunction*[3] ermöglicht die Wiederverwendung von Code im Kontext der Vererbung (s. Kapitel 3.2) und führt zu dem angenehmen Nebeneffekt, dass der erzeugte Code auch für den Menschen verständlich und lesbar bleibt. Die Funktionen `loop` und `new` sind generisch und für jedes Template gleich. Sie tauchen nur einmal in der Übersetzung eines ObjectCurry-Programms auf. Die übrigen Funktionen werden für jedes im Programm definierte Template individuell erzeugt.

Das Senden von Nachrichten an ein Objekt geschieht über den Constraint `sendMsg`:

```
sendMsg msg obj = send (MSG msg) obj
```

3.1.4. Self

Innerhalb eines Templates sind neben allen global deklarierten Funktionen, Datentypen, Konstruktorfunktionen und Nachrichten auch die im Template beschriebenen Attribute sichtbar. Alle diese Bezeichner können innerhalb der Zuweisungen und Constraints der Methodenrümpfe verwendet werden. Die Attribute sind mit den Werten des jeweiligen

[2] *template* steht jeweils für den Namen des Templates
[3] *constructorFunction* ist der Name der im Template definierten Konstruktorfunktion

Objekts belegt. Darüber hinaus liegen auch die Parameter der Nachrichten innerhalb des Sichtbarkeitsbereichs des Methodenrumpfs. Möchten wir jedoch auf das Objekt selbst innerhalb einer Methode zugreifen, so ist dies bisher nur über einen Umweg möglich: Das Objekt selbst muss über einen Parameter der Nachricht mitgeschickt werden. Dadurch gelangt es in den Sichtbarkeitsbereich des Methodenrumpfs und kann dort verwendet werden, z. B. um ihm, also sich selbst, weitere Nachrichten zu schicken, oder um es an andere Funktionen oder Objekte weiterzureichen. Eine andere Möglichkeit, das Objekt selbst innerhalb seiner eigenen Methoden verfügbar zu machen, besteht darin, es in einem zusätzlichen Attribut abzulegen, das unmittelbar nach der Erzeugung des Objekts mit dem Objekt selbst belegt wird. Beide Möglichkeiten lassen sich in ObjectCurry explizit anwenden. Bequemer ist es jedoch, einem Objekt den Zugriff auf sich selbst zu gestatten, indem ein neuer Bezeichner `self` implizit in jeder Methode zur Verfügung gestellt wird. Dies kann auf die zwei beschriebenen Weisen geschehen:

1. Der Port auf das Objekt wird als zusätzlicher Parameter an jede Nachricht angehängt. Implizit lässt sich dies durch eine Modifikation der `send`- und Action-Funktionen erreichen:

   ```
   data ModifiedMsg a = ModifiedMsg Msg a
   modifiedSend msg obj = send (ModifiedMsg msg obj) obj

   ...
   counterAction (x) (ModifiedMsg Inc self)
     = State (x + 1)
   ...
   ```

2. Der Port auf das Objekt wird in einem zusätzlichen Attribut abgelegt, auf das in allen Methoden zugegriffen werden kann. Um dieses Verfahren zu realisieren, muss zunächst die `new`-Funktion erweitert werden:

   ```
   new constructor port =
     let stream free in
       openPort port stream &> constructor port stream
   ```

   Zusätzlich zu dem Nachrichtenstrom `stream` wird auch der Port `port` an die Konstruktorfunktion `constructor` übergeben. Diese reicht den Port an die Schleife `loop` weiter, die die Nachrichten einzeln aus dem Strom ausliest und nun zusammen mit dem Port als zusätzlichen Parameter `self` an die zugehörige Action-Funktion weitergibt.

   ```
   loop eval rigid
   loop state action self (m:ms) =
     continuation nextState self ms
     where
       nextState = action state self m
       continuation (State ns) self ms = loop ns action self ms
       continuation Final _ _ = success
   ```

```
counter eval rigid
counter init self = loop (counterInitState (init)) counterAction self

...
counterAction (x) self (MSG Stop) = Final
counterAction (x) self (MSG Inc)
  = State (x + 1)
...
```

Auf diese Weise kann ein Objekt über `self` in jeder Methode auf sich selbst zugreifen.

Während die zweite Möglichkeit den Zustandsraum jedes Objekts vergrößert, verlängert das erste Verfahren jede Nachricht, die an das Objekt verschickt wird. Da Nachrichten auch über ein Netzwerk an Objekte auf anderen Computern geschickt werden können (s. Kapitel 4.1.2), ist der zusätzliche Platz- und Zeitbedarf in diesem Fall größer. Daher wurde in ObjectCurry die zweite Möglichkeit implementiert.

3.2. Vererbung

Objektorientierte Programmiersprachen erlauben es in der Regel, Klassen nicht nur als Schablone zur Definition von Objekten, sondern auch zur Definition weiterer Klassen zu verwenden ([CW97]). Die Vererbung von Klasseneigenschaften ermöglicht es, vorhandenen Code wiederzuverwenden. Eine Klasse erbt, wenn ihr eine andere Klasse als **Oberklasse** zugeordnet wird. Die erbende Klasse wird **Unterklasse** genannt. Vererbt werden sowohl Instanzvariablen als auch Methoden. Der Satz an geerbten Instanzvariablen kann durch die Unterklasse **erweitert** werden, ebenso lassen sich neue Methoden **definieren**. Geerbte Methoden können **redefiniert** werden, d. h. es wird eine neue Verhaltensweise festgelegt, wie ein Objekt der Unterklasse auf eine bestimmte Nachricht reagiert.

Einige objektorientierte Programmiersprachen gestatten darüber hinaus auch **Mehrfachvererbung**. Hierbei werden einer Klasse mehrere Oberklassen zugeordnet. Dies führt zu Problemen, wenn Attribute oder Methoden der Oberklassen identische Namen tragen. Deshalb müssen die jeweiligen Programmiersprachen zusätzliche Konstrukte zum Lösen dieser Probleme bereitstellen. Doch auch wenn von diesen Problemen abgesehen wird, ist Mehrfachvererbung nur in relativ wenigen Situationen wirklich erforderlich, um ein Problem der realen Welt objektorientiert zu modellieren. Daher wurde in ObjectCurry auf die Implementierung der Mehrfachvererbung verzichtet.

Auch in ObjectCurry ist es möglich, Templates auf der Grundlage anderer Templates zu definieren. Zu diesem Zweck ordnet man einem Template S ein anderes Template V zu, von welchem es alle Attribute und Methoden erbt. Ein Template S, welches von einem Template V erbt, nennen wir **Subtemplate von V** und V **Vorfahre von S**. Ein Subtemplate von V versteht alle Nachrichten von V.

Ist der Vorfahre V eines Templates S ebenfalls ein Subtemplate eines dritte Templates O, so erbt S indirekt auch alle Methoden von O. Wir nennen O in diesem Fall ebenfalls einen Vorfahren von S, V hingegen bezeichnen wir auch als **direkten Vorfahren von**

$\boldsymbol{S}$. Darüber hinaus kann ein Template auch als sein eigener Vorfahre aufgefasst werden. Deshalb nennen wir die anderen Vorfahren **echte Vorfahren**. Wir bezeichnen die **Relation der direkten Vorfahren** eines Programms P mit $\mathcal{H}_P$: $(S, V) \in \mathcal{H}_P$, falls V direkter Vorfahre von S in dem Programm P.

In einem Subtemplate S können weitere Attribute definiert werden, die den Zustandsraum eines Objekts dieses Templates erweitern. Der Zustandsraum einer Instanz von S ist das kartesische Produkt aus dem Zustandsraum eines Objekts des Vorfahren von S und den in S zusätzlich definierten Attributen.

Das Verhalten von Objekten eines Subtemplates auf Nachrichten kann ebenfalls verändert werden. Insgesamt gibt es drei Arten, wie Methoden in Subtemplates bestimmt werden:

1. Methoden werden geerbt: Ein Objekt eines Subtemplates erbt jede Methode seines Vorfahren, sofern sie nicht redefiniert wird.
2. Methoden können redefiniert werden: Innerhalb des Subtemplates werden neue Reaktionen auf solche Nachrichten festgelegt, die bereits von Objekten des Vorfahren verstanden werden.
3. Es können neue Methoden definiert werden: Ein Objekt eines Subtemplates S versteht dann mehr Nachrichten als ein Objekt des Vorfahren von S.

In ObjectCurry werden Subtemplates durch das reservierte Schlüsselwort `extends`, gefolgt von dem Namen des direkten Vorfahren gekennzeichnet.

Als Beispiel möchten wir das Zählerobjekt um eine obere Schranke erweitern. Sobald der Zähler diese Schranke erreicht hat, bleibt die Nachricht `Inc` ohne Reaktion. Zusätzlich führen wir eine neue Nachricht `SetMax` ein, über die die Schranke nachträglich verändert werden kann. Das folgende Template ermöglicht die Erzeugung solcher Objekte:

```
template MaxCounter extends Counter =
constructor
  maxCounter init maxInit = counter init
                            max := maxInit
methods
  Inc           = x   := (if x < max then x+1 else x)
  SetMax newMax = max := newMax
```

Die Definition eines Subtemplates unterscheidet sich syntaktisch von den bisher vorgestellten Templates durch das reservierte Schlüsselwort `extends`, gefolgt von dem Namen des Vorfahren. Dieses zeigt an, dass es sich bei `MaxCounter` um ein Subtemplate handelt. Der Vorfahre ist das Template `Counter`.

Ein Subtemplate erbt alle Attribute seines Vorfahren. `MaxCounter` erbt demnach das Attribut `x` aus `Counter`. Im Initialisierungsteil des Subtemplates können weitere Attribute eingeführt werden. Ein MaxCounter-Objekt besitzt zusätzlich zu `x` das neu eingeführte Attribut `max`.

Der Initialwert der neu definierten Attribute wird wie gewohnt durch die Konstruktorfunktion festgelegt. Auch die geerbten Attribute müssen mit einem Initialwert belegt werden, damit jede Instanz einen eindeutigen Startzustand besitzt. Aus diesem Grund muss die erste Zeile der Konstruktorfunktion eines Subtemplates einen Aufruf der Konstruktorfunktion ihres Vorfahren enthalten. Im Beispiel erhält das geerbte Attribut `x` somit den Initialwert `init`.

Eine andere Möglichkeit wäre gewesen, die geerbten Attribute ebenfalls durch die Angabe von Zuweisungen zu initialisieren. Dies wäre bei vielen geerbten Attributen jedoch weitaus umständlicher.

Betrachten wir nun die Methoden. Das Verhalten eines MaxCounter-Objekts beim Empfang der Nachricht `Inc` unterscheidet sich von dem eines Counter-Objekts: Sobald der Zähler den Maximalwert erreicht hat, darf er nicht mehr weiter inkrementiert werden. Aus diesem Grund muss die Methode `Inc` redefiniert werden. Eine Methode lässt sich redefinieren, indem eine neue Methode mit demselben Namen definiert wird. Diese ersetzt dann die Definition des Vorfahren.

Innerhalb des Rumpfs einer Methode sind sowohl die im Template neu definierten, als auch die vom Vorfahren geerbten Attribute sichtbar. In der Definition von `Inc` innerhalb von `MaxCounter` kann daher `x` wie in `Counter` verwendet werden.

Viele objektorientierte Sprachen verfügen über Mechanismen zum **Verbergen** von einzelnen Bezeichnern. In Java wird dies beispielsweise mittels **Zugriffsangaben** wie `private` oder `public` realisiert. Eine andere Möglichkeit besteht darin, explizit anzugeben, welche Attribute und Methoden geerbt werden sollen, so z. B. realisiert in Haskell++.

In ObjectCurry ist jedes Attribut nach außen vollständig gekapselt. Nur innerhalb des Templates, in dem ein Attribut definiert wurde, sowie in dessen Subtemplates ist das Attribut sichtbar. Ein weiteres Einschränken der Sichtbarkeit von einzelnen Bezeichnern innerhalb der Subtemplates ist nicht vorgesehen, ließe sich aber ohne großen Aufwand durch eine zusätzliche semantische Prüfung im Compiler bei Bedarf realisieren.

Die Definition neuer Methoden erfolgt in einem Subtemplate in der gleichen Weise wie in einem Template ohne Vorfahren. In unserem Beispiel wird die Methode `SetMax newMax` neu eingeführt. Durch Schicken der Nachricht `SetMax` m an ein MaxCounter-Objekt wird dessen obere Schranke auf den neuen Wert m gesetzt, und gegebenenfalls der Zähler auf diesen Wert erniedrigt.

Ein Subtemplate erbt jede Methode seines Vorfahren, sofern es sie nicht redefiniert. Jedes MaxCounter-Objekt erbt daher die Methoden `Get v` und `Set v`, d. h. ein MaxCounter-Objekt reagiert auf die Nachrichten `Get v` und `Set v` genau wie ein Counter-Objekt: `Set v` setzt den internen Zähler auf den Wert `v`, und `Get v` bindet die freie Variable `v` an den Wert des Zählers.

3.2.1. Übersetzung von Subtemplates

Die Übersetzung eines Subtemplates S mit dem Vorfahren V nach Curry unterscheidet sich in zwei Punkten von der Übersetzung eines Templates ohne Vorfahren:

1. Ein Teil des Startzustands wird durch den Vorfahren bestimmt. Die Initialisierungsfunktion von S ruft dazu die Initialisierungsfunktion von V auf.
2. Eine Instanz von S muss auch auf die Nachrichten reagieren, zu denen keine Methode in S definiert bzw. redefiniert wird, sondern deren Reaktionen durch V festgelegt sind.

Initialisierung

Die Initialisierungsfunktion liefert den Startzustand einer neuen Instanz eines Templates. Da dieser bei Subtemplates auch durch den Vorfahren bestimmt ist, ermittelt die Initialisierungsfunktion eines Subtemplates S zunächst den Startzustand einer potentiellen Instanz des Vorfahren V. Anschließend wird dieser um die Initialwerte der neu definierten Attribute erweitert. Die Initialisierungsfunktion von MaxCounter hat folgendes Aussehen:

```
maxCounterInitState (init,maxInit) =
  let (r_x) = counterInitState (init)
  in (r_x,maxInit)
```

Die Konstruktorfunktion lautet wie folgt:

```
maxCounter init maxInit self =
  loop (maxCounterInitState (init,maxInit)) maxCounterAction self
```

Neue und redefinierte Methoden

Durch das Redefinieren der Methode eines Vorfahren wird diese vollständig überschrieben, d. h. diese hat keinen Einfluss auf das Subtemplate. Daher können neue und redefinierte Methoden von Subtemplates genauso wie Methoden von Templates ohne Vorfahren, übersetzt werden.[4] Analog zu Kapitel 3.1.3 ergibt sich damit:

```
maxCounterAction (x,max) self (MSG Inc)
  = State ((if x < max then x + 1 else x),max)

maxCounterAction (x,max) self (MSG (SetMax newMax))
  = State (x,newMax)
```

Geerbte Methoden

Anders verhält es sich bei geerbten Methoden. Die Reaktion auf eine Nachricht ist in diesem Fall in dem Vorfahren des Templates zu finden und muss zusätzlich zu den neu definierten und redefinierten Methoden in die Action-Funktion aufgenommen werden. Hierfür bieten sich uns zwei Möglichkeiten:

1. Copy & Paste

[4] außer bei der Verwendung von `super`, s. Kapitel 3.2.4

2. Delegation

Bei **Copy & Paste** wird der Code der geerbten Methode kopiert und in das Subtemplate eingefügt. Diese Möglichkeit besitzt den Nachteil, dass der generierte Code besonders bei großen Methodenrümpfen unnötig aufgebläht wird. Bei der **Delegation** hingegen wird in der Implementierung der Code der geerbten Methode aufgerufen. Wir machen in ObjectCurry von der zweiten Weise Gebrauch.

In der Implementierung geben wir auch für jede geerbte Methode eine definierende Gleichung der Action-Funktion an. Dabei bedienen wir uns jedoch der Action-Funktion des direkten Vorfahren. Dieser übergeben wir neben `self` und der jeweiligen Nachricht die Werte der Attribute, die geerbt werden, also die Projektion des aktuellen Zustands auf die geerbten Attribute. Als Ergebnis erhalten wir die Projektion des gesuchten Nachfolgezustands auf die geerbten Attribute. Da durch den Aufruf einer geerbten Methode die neu eingeführten Attribute niemals verändert werden, erhalten wir den gesamten Nachfolgezustand, indem wir an die Projektion die neu eingeführten Attribute anhängen.

```
maxCounterAction (x,max) self (MSG (Get v))
  = let State (r_x) = counterAction (x) self (MSG (Get v))
    in State (r_x,max)

maxCounterAction (x,max) self (MSG (Set v))
  = let State (r_x) = counterAction (x) self (MSG (Set v))
    in State (r_x,max)
```

3.2.2. Subtypen

Grundsätzlich existieren in objektorientierten Systemen zwei konzeptuelle Sichtweisen der Vererbung. Zum einen lassen sich Klassen durch Vererbung erweitern bzw. modifizieren. Hier spricht man von der **Modulsichtweise**. Zum anderen gibt es die so genannte **Typsichtweise**: Durch Vererbung wird ein Untertyp B (speziellerer Typ) der Oberklasse A definiert, d. h. B-Objekte lassen sich überall dort einsetzen, wo eigentlich A-Objekte verlangt ([Han98]) werden. Bsp.: Ein MaximumCounter-Objekt lässt sich überall dort einsetzen, wo eigentlich ein Counter-Objekt verlangt wird.

Objektorientierte Programmiersprachen verfügen in der Regel über so genannten Subtyppolymorphismus. Erst dieser ermöglicht die Flexibilität, die diese Klasse von Sprachen auszeichnet. Der Subtyppolymorphismus erlaubt die Verwendung der Vererbung im Sinne der Typsichtweise: Ein Objekt einer Unterklasse von T kann an jeder Stelle stehen an, an der ein Objekt der Klasse T verlangt wird. Dies macht Sinn, da das Objekt der Unterklasse alle Nachrichten versteht, die eine Instanz von T versteht.

Wir möchten in ObjectCurry erlauben, Datenstrukturen mit Elementen verschiedener Klassen zu erzeugen, beispielsweise eine Liste mit Nachrichten verschiedener Klassen:

```
[Inc,MaxInc]
```

Diese Liste wird als Liste von Nachrichten des Typs `Message MaxCounter` aufgefasst, auch wenn sie eine Nachricht des Typs `Message Counter` enthält, denn nur Instanzen von

`MaxCounter` können alle Nachrichten dieser Liste verstehen.

Genau umgekehrt verhält es sich mit Objekten. Sei `counterObj` eine Instanz der Klasse `Counter` und `maxCounterObj` eine Instanz von `MaxCounter`. Die Liste

```
[counterObj,maxCounterObj]
```

wird dann als Liste von Counter-Objekten aufgefasst, da alle Objekte der Liste maximal die in `Counter` definierten Nachrichten verstehen.

In Curry steht Subtyppolymorphismus jedoch (s. Kapitel 5.1) nicht zur Verfügung. Uns stehen die folgenden Möglichkeiten zur Verfügung, mit diesem Problem in der Übersetzung eines ObjectCurry-Programms umzugehen:

1. Wir verzichten generell auf Subtyppolymorphismus in ObjectCurry. Ein Template kann zwar Attribute und Methoden von einer anderen Klasse erben, die obigen Beispiele aber sind unzulässig.
2. Wir verwenden Coercions wie in O'Caml (s. Kapitel 1.4.3), um den Typ eines Objekts oder einer Nachricht manuell zu konvertieren, wenn dies nötig ist. In diesem Fall würde bei der Bildung der Liste `[Inc, MaxInc]` die Anpassung des Typs von `MaxInc` erfordern. Eine mögliche Schreibweise wäre

   ```
   [Inc, MaxInc : MaxCounter :> Counter]
   ```

3. Als letzte Möglichkeit können wir Polymporphimus ohne Typprüfung verwenden. In diesem Fall geht man davon aus, dass alle Klassen denselben Typ besitzen. Dabei entsteht das Problem, dass nun Nachrichten an Objekte geschickt werden können, die diese nicht verstehen, ohne dass der Typchecker diesen Fehler erkennt.

Bei der Übersetzung von ObjectCurry-Programmen nach Curry machen wir von der letzten Möglichkeit Gebrauch. Der anschließende Lauf eines Curry-Compilers entdeckt daher keine Verletzungen der Konformität. Wir verwenden einen zusätzlichen Typchecker, der bereits auf ObjectCurry-Ebene alle Typfehler erkennt. Nur typkorrekte Programme werden nach Curry übersetzt. Das dem Typchecker zugrunde liegende Typsystem wird ausführlich in Kapitel 5 behandelt.

3.2.3. Virtuelle Methoden und Dynamisches Binden

Das Senden von Nachrichten geschieht in den meisten objektorientierten Programmiersprachen in der Regel durch den Aufruf der entsprechenden Methode, denn Methoden sind als Funktionen oder Prozeduren implementiert. Betrachten wir den folgenden Fall: Eine Klasse A definiert zwei Methoden M_1 und M_2. In der Methode M_1 wird die Methode M_2 aufgerufen. Eine Klasse B erbt von A, redefiniert aber die Methode M_2. Nun erzeugen wir ein Objekt O als Instanz der Klasse B und rufen die geerbte Methode M_1 dieses Objekts auf. Für den darin enthaltenen Aufruf der Methode M_2 gibt es nun zwei Möglichkeiten: Entweder wird die Implementierung der Klasse A verwendet, da M_1 in A

definiert ist, oder man verwendet die Implementierung von M_2 der Klasse B, da es sich bei dem aktuellen Objekt um eine Instanz von B handelt. Im letzteren Fall spricht man von **virtuellen Methoden**.

Die Implementierung virtueller Methoden erfordert in konventionellen objektorientierten Programmiersprachen einen Mechanismus, der **dynamisches Binden** genannt wird: Bei einem Aufruf einer Methode wird erst zur Laufzeit, also dynamisch, ermittelt, welche Implementierung verwendet wird.

In ObjectCurry haben wir es immer mit „virtuellen Methoden" zu tun. Trotzdem brauchen wir dem Thema „dynamisches Binden" keine weitere Beachtung zu schenken. Dazu betrachten wir das folgende Beispiel:

```
template A =
constructor
  a = x := 0
methods
  M1    = send M2 self
  M2    = x := 1
  Get v = x =:= v

template B extends A =
constructor
  b = a
methods
  M2 = x := 42

test | new b o & (sendMsg M1 o &> sendMsg (Get v) o) = v
  where o,v free
```

Die Auswertung von `test` liefert den Wert 42. In Curry entspricht das Aufrufen einer Methode dem Senden einer Nachricht an `self`. Das Objekt selbst ist daher für die Reaktion auf diese Nachricht verantwortlich und verwendet die Implementierung seines Templates.

3.2.4. Aufruf verborgener Methoden

Mitunter ist es nötig, innerhalb eines Objekts auf Methoden des Vorfahren zurückzugreifen. Handelt es sich dabei um eine nicht redefinierte Methode, kann dies durch das Senden einer Nachricht an `self` erreicht werden. Wurde die Methode jedoch im Subtemplate redefiniert, so ist die ursprüngliche Implementierung verborgen.

Aus diesem Grund stellt ObjectCurry wie die meisten objektorientierten Programmiersprachen eine zusätzliche Anweisung zur Verfügung, die wir `super` nennen. Enthält die rechte Seite einer Methode die Anweisung `super m`, so werden die folgenden Aktionen ausgeführt:

- Die Constraints der Methode `m` werden gelöst.
- Der Nachfolgezustand der geerbten Attribute wird aus den in `m` definierten Zuweisungen berechnet.

- Alle Zuweisungen an geerbte Attribute innerhalb des Methodenrumpfs werden ignoriert.

Als Beispiel möchten wir in `MaxCounter` eine zusätzliche Methode `Inc2` definieren, die den Zähler ohne Beachtung der Schranke durch Zugriff auf die `Inc`-Methode des Vorfahren `Counter` inkrementiert.

```
Inc2 = super Inc
```

Die entsprechende Gleichung der Action-Funktion verwendet eine ähnliche Technik wie die Gleichungen geerbter Methoden:

```
maxCounterAction (x,max) self Inc2
  = let State (s1_x) = counterAction (x) self Inc
    in State (s1_x,max)
```

3.3. Zusätzliche Sprachelemente

Zusätzlich zu der bisher vorgestellten Syntax bietet ObjectCurry weiteren syntaktischen Zucker, der die Definition von Templates vereinfacht. Durch die Angabe von Bedinungen lässt sich beispielsweise die Methode `Inc` des MaxCounter-Templates besser lesbar definieren:

```
  Inc | (x <  max)=:=True = x := x+1
      | (x >= max)=:=True = x := x
```

3.4. Formale Syntax

Bisher wurde die Syntax von ObjectCurry informell mit Hilfe von Beispielen eingeführt. Formal lässt sich die Syntax durch die Angabe einer kontextfreien Grammatik definieren. Eine Grammatik, die die Syntax von Curry beschreibt, findet sich in [Han00a]. Da ObjectCurry lediglich eine Erweiterung der Sprache Curry darstellt, reicht es aus, diese Grammatik um zusätzliche EBNF-Regeln zu erweitern. Tabelle 3.1 auf Seite 37 beinhaltet diese Regeln.

Die Regel *BlockDeclaration* der Curry-Grammatik muss um eine zusätzliche Alternative ergänzt werden, denn auf Toplevel ist in ObjectCurry neben der Deklaration von abstrakten Datentypen und Funktionen auch die Deklaration von Templates möglich. Der Aufbau einer Templatedeklaration wird durch die Regel *TemplateDecl* beschrieben.

3.5. Polymorphe Templates

In der vorgestellten Sprache ObjectCurry können auch polymorphe Templates definiert werden. Ein Template ist polymorph, wenn der Typ mindestens eines Attributs zur Compilezeit nicht festgelegt ist, sondern aus einer Typvariable besteht. Bei der Bildung einer Instanz eines solchen Typs muss jedoch der Typ aller Attribute feststehen.

BlockDecl.	::=	*DataDeclaration*	
	|	*FunctionDeclaration*	
	|	*TemplateDecl*	
TemplateDecl	::=	template *TypeConstrID*	
		constructor [*Signature*]	
		= *FunLHS Assignments*	
		methods *MethodDecls*	
	|	template *TypeConstrID* extends *TypeConstrID*	
		constructor [*CSignature*]	
		FunLHS = *FatherCall Assignments*	
		methods *MethodDecls*	
Assignments	::=	{ $Assignment_1$; ... ; $Assignment_n$ }	$n \geq 0$
Assignment	::=	*VariableID* := *Expression*	
FatherCall	::=	*FunctionName* $BasicExpr_1$ ... $BasicExpr_n$	$n \geq 0$
MethodDecls	::=	{ $MethodDecl_1$; ... ; $MethodDecl_n$ }	$n \geq 1$
MethodDecl	::=	[*MSignature*] *MethodLHS MethodRule*	
	|	[*MSignature*] *MethodLHS* = *AssgOrExprs*	
MethodLHS	::=	*DataConstrID* $SimplePattern_1$ ... $SimplePattern_n$	$n \geq 0$
MethodRule	::=	| *Expression* = *AssgOrExprs* [*MethodRule*]	
AssgOrExprs	::=	{ $AssgOrExpr_1$; ... ; $AssgOrExpr_n$ }	$n \geq 1$
AssgOrExpr	::=	*Assignment*	
	|	*Expression*	
	|	sendMsg *Expression* self	
	|	super *DataConstrID* $BasicExpr_1$ ... $BasicExpr_n$	$n \geq 0$
CSignature	::=	*FunctionName* :: *TypeExpr*	
MSignature	::=	*DataConstrID* :: *TypeExpr*	

Tabelle 3.1.: Syntax von ObjectCurry

Im Typsystem in Kapitel 5 erlauben wir jedoch keine polymorphen Templates.

Polymorphe Objekte verbieten wir ganz: Ein polymorphes Objekt ist ein Objekt, das ein Attribut besitzt, dessen Typ zum Zeitpunkt der Objekterzeugung nicht festgelegt ist.

4. Verteilte Objektorientierte Programmierung

Verteilte Anwendungen erfordern die Kommunikation zwischen verschiedenen Prozessen, die auf unterschiedlichen Rechnern laufen können. Auch bei der Erstellung von verteilten Anwendungen möchte man auf einem möglichst abstrakten Level programmieren. In der Industrie eingesetzte Techniken wie JavaRMI (Java Remote Method Invocation) [rmi00] oder Corba [cor] zeigen, dass Objekte eine in der Praxis funktionierende und leicht anzuwendende Möglichkeit darstellen, verteilte Anwendungen zu erstellen: Die Methoden eines Objekts, das auf einem bestimmten Rechner erzeugt wurde, werden der Außenwelt sichtbar gemacht. Über diese haben dann auch externe Prozesse eine Zugriffsmöglichkeit auf ein solches Objekt.

Mit JavaRMI beispielsweise lassen sich einzelne Methoden eines Objekts veröffentlichen. Ein Prozess, der auf ein externes Objekt zugreifen möchte, erzeugt zunächst ein Skelett-Objekt, das dieselben Methoden enthält, wie das veröffentlichte Objekt. Ruft er eine dieser Methoden auf, so schickt das Skelett-Objekt eine Nachricht an den Rechner, auf dem das veröffentlichte Objekt liegt. Dort wird die entsprechende Methode dieses Objekt aufgerufen, und das Ergebnis zurück an das Skelett-Objekt geschickt, welches es als Funktionsergebnis an den ursprünglichen Aufrufer zurückliefert. Die Generierung der dazu notwendigen Schnittstellen und der Skelett-Klasse übernimmt ein spezieller Compiler.

Da Nachrichten in ObjectCurry Elemente eines ADT sind und direkt über Ports verschickt werden können, ermöglichen externe Ports die Kommunikation zwischen verschiednen Rechnern.

Im folgenden Abschnitt gehen wir zunächst auf die Funktionen ein, mit denen externe Ports in Curry erzeugt und verwendet werden, und stellen dann eine Möglichkeit vor, diese zur Erzeugung verteilter Objekte einzusetzen. Mit einem Ansatz zur Synchronisation von Nachrichten beschäftigt sich Kapitel 4.2.

4.1. Externe Ports und Verteilte Objekte

4.1.1. Externe Ports

In Curry existieren zwei Funktionen zur Erzeugung und Verwendung externer Ports. Ein externer Port unterscheidet sich von einem internen Port dadurch, dass auch externe Prozesse Nachrichten an diesen schicken können.

Die Funktion `openNamedPort` n erzeugt einen neuen **externen Port** mit dem Namen n und gibt den eingehenden Nachrichtenstrom zurück. Wird diese Funktion auf einem Rechner mit dem Namen `m` ausgeführt, so wird der so erzeugte Port unter dem Namen `n@m` veröffentlicht [Han99]. Die Funktion `openNamedPort` verändert die Außenwelt, besitzt also einen Seiteneffekt. Daher wurde sie als I/O-Aktion definiert. `openNamedPort` besitzt den Typ `String -> IO [a]`. Dabei ist `a` der Nachrichtentyp des Ports.

Auf Client-Seite liefert die I/O-Aktion `connectObject` n den externen Port mit dem Namen n zurück. Auf diese Weise kann sich ein Client mit einem Port des Namens n verbinden und anschließend mittels `send` Nachrichten an diesen versenden.

4.1.2. Verteilte Objekte

Mit Hilfe externer Ports lassen sich verteilte Objekte erzeugen und benutzen. Diese werden wie externe Objekte über einen symbolischen Namen der Außenwelt zur Verfügung gestellt. Ein Client auf einem anderen Rechner kann sich mit einem externen Objekt verbinden und Nachrichten an dieses senden.

Im Gegensatz zu Techniken wie JavaRMI oder Corba benötigen wir in ObjectCurry keine zusätzlichen Schnittstellen, da das Senden einer Nachricht nicht durch den Aufruf einer Methode realisiert ist.

Zum Erzeugen eines externen Objekts führen wir die Nachricht `newNamedObject` ein. Dieser übergeben wir den Namen, unter dem das Objekt veröffentlicht werden soll, und die Konstruktorfunktion, die das Objekt erzeugt. `newNamedObject` ist folgendermaßen implementiert:

```
newNamedObject name constructor =
  openNamedPort name >>=
  \stream -> putStr name >> connectPort name >>=
  \port -> solve (constructor port stream)
  where solve x | x = done
```

Ein Aufruf von `newNamedObject name constructor` erzeugt zunächst einen externen Port mit dem Namen `name`. `openNamedPort` liefert den Strom der eingehenden Nachrichten, der an die Konstruktorfunktion `constructor` übergeben wird. Zusätzlich benötigt diese auch den Port selbst, um einem Objekt das Schicken von Nachrichten an sich selbst zu ermöglichen (s. Kapitel 3.1.4). Zu diesem Zweck wird über `connectPort` eine Verbindung zu dem gerade erzeugten externen Port hergestellt und als `port` ebenfalls an die Konstruktorfunktion übergeben.

Über die Funktion `connectObject` kann sich ein Programm mit einem durch `newNamedObject` erzeugten Objekt auf einem anderen Rechner verbinden. Nachrichten an das Objekt werden wie gewohnt mit `sendMsg` an das Objekt geschickt. Die Implementierung von `connectObject` ist einfach, denn ein Objekt ist aus Sicht eines externen Senders nichts weiter als ein Port:

```
connectObject n = connectPort n
```

Das folgende Beispiel zeigt einen Server, der ein Counter-Objekt erzeugt und externen Sendern zur Verfügung stellt, sowie einen Client, der sich mit diesem Objekt verbindet und die Nachrichten `Inc` und `Get v` an dieses sendet.

Bsp. 4.1.1

```
server = newNamedObject "counter" (counter 42)

client = connectObject "counter@agamemnon" >>= h
  where
   h object | sendMsg Inc object & sendMsg (Get v) object = putStr (show v)
   v free
```

Durch Senden der Nachricht `Stop` an das Counter-Objekt wird der Server-Prozess beendet. Das Beispiel zeigt jedoch, dass beim Einsatz von verteilten Objekten, das Senden der `Stop`-Nachricht (vgl. **??**, **??**) nicht unbedingt erforderlich ist. Ein Objekt, das die Nachricht `Stop` nicht erhält, wartet endlos auf weitere Nachrichten.

4.2. Synchronisation

In verteilten Anwendungen kann es zu Synchronisationsproblemen kommen, da Abhängigkeiten zwischen verschiedenen Prozessen bestehen und Prozesse gegebenenfalls auf andere Prozesse warten müssen. In verteilten Programmiersprachen existieren eine Vielzahl von Sprachkonstrukten, um Prozesse zu synchronisieren, wie z. B. **Semaphore**, **Monitore** oder **Rendevouz-Mechanismen**. Während Semaphore auf einem niedrigen Abstraktionsgrad anzusiedeln sind, stellen Monitore und Rendevouz-Mechanismen hohe Konzepte zur Synchronisation dar [Han98]. Aus der Menge von Synchronisationsmechanismen und -techniken haben wir das Konzept des Monitors ausgewählt, da dieses sehr elegant ist [Han98] und sich mit einfachen Mitteln in ObjectCurry integrieren lässt.

Im folgenden Abschnitt stellen wir vor, was ein Monitor ist, und wie er sich in ObjectCurry integrieren lässt. Abschnitt 4.2.2 beschreibt die Implementierung des Monitorkonzepts bei der Übersetzung von ObjectCurry nach Curry.

4.2.1. Synchronisation durch Monitore

Ein Monitor ist ein Konstrukt zur Synchronisation von Prozessen: Er kapselt lokale Daten und Operationen auf diesen Daten und stellt einen Synchronisationsmechanismus zur Verfügung: „Ein Monitor besteht aus lokalen Daten, Operationen auf diesen Daten, Initialisierungsteil und einem Mechanismus zum Suspendieren und Aufwecken von Operationen“ [Han98]. Dabei erfolgt der Zugriff auf die lokalen Daten ausschließlich über die zur Verfügung gestellten Operationen, und nur ein Prozess kann gleichzeitig in den Monitor eintreten. Der Synchronisationsmechanismus eines Monitors basiert auf den folgenden zwei Anweisungen:

- `delay(r)` fügt den aufrufenden Prozess in die Warteschlange `r` ein.
- `continue(r)` aktiviert einen Prozess aus der Warteschlange `r`. Befindet sich kein Prozess in der Warteschlange, bleibt die Anweisung `continue` ohne Effekt. [1]

[1] A. Silberschatz [SG94] verwendet für diese Funktionen statt `delay` den Namen `wait` und statt `continue` den Namen `signal`. Die Sprache Java, in der ein Monitor-ähnliches Konzept implementiert wurde, verwendet die Bezeichnungen `wait` und `notify`.

Bis auf den Mechanismus zur Suspension und zum Aufwecken von Operationen entspricht ein Monitor exakt einem Objekt in ObjectCurry: Ein Objekt besitzt lokale Daten (Attribute), einen Initialisierungsteil (Konstruktorfunktion) sowie Operationen (Methoden). Der Zugriff erfolgt ausschließlich über die Operationen, nämlich indirekt durch das Senden von Nachrichten, und nur ein Prozess kann gleichzeitig in den Monitor eintreten, da ein Objekt als rekursive Schleife implementiert ist. Unsere Aufgabe besteht also lediglich darin, einen Mechanismus zur Synchronisation zur Verfügung zu stellen. Wir erweitern die Syntax von ObjectCurry dahingehend, dass auf der rechten Seite einer Methode neben Zuweisungen und Constraints auch die oben beschriebenen Anweisungen `delay r` und `continue r` auftreten können, wobei `r` eine Warteschlange ist. Die Funktion `emptyQueue` liefert eine leere Warteschlange.

Als Beispiel betrachten wir einen einelementigen Puffer, den wir in Form eines Objekts implementieren. Dieser versteht die Nachrichten `Put a` und `Get a`, mit denen ein Element in den Puffer abgelegt bzw. aus diesem geholt werden kann. Das Attribut `x` besitzt den Wert `Just y`, falls der Wert `y` in den Puffer gelegt wurde, oder `Nothing`, falls der Puffer leer ist. Die Nachricht `Put a` kann nur erfolgreich verarbeitet werden, wenn der Puffer leer ist; umgekehrt liefert `Get a` nur dann eine Antwort (durch Binden von `a`), falls der Puffer einen Wert enthält. Zur Synchronisation benötigen wir zwei Warteschlangen `r1` und `r2`: In die erste fügen wir über `delay` diejenigen Prozesse ein, die eine `Put`-Nachricht gesendet haben, obwohl der Puffer voll war, und in die zweite diejenigen Prozesse, die bei leerem Puffer eine `Get`-Nachricht geschickt haben. Ein Template zur Erzeugung eines solchen Puffers lässt sich wie folgt definieren:

```
template Buffer =
constructor
  buffer i =
    x := Just i
    r1 := emptyQueue
    r2 := emptyQueue
methods
  synchronized Put a
    | x=:=Nothing = x := Just a     -- Puffer leer: Element speichern,
                    continue r2     --   Prozess aus r2 aufwecken
    | x=:=Just x2 = delay r1        -- Puffer voll: Prozess suspendieren
    where x2 free
  synchronized Get a
    | x=:=Nothing = delay r2        -- Puffer leer: Prozess suspendieren
    | x=:=Just x2 = a =:= x2        -- Puffer voll: Element auslesen,
                    x := Nothing    --   Puffer leeren,
                    continue r1     --   Prozess aus r1 aufwecken
    where x2 free
```

Um Overhead in der Implementierung zu vermeiden, sind die Anweisungen `delay r` bzw. `continue r` nur in solchen Methoden erlaubt, die durch voranstellen des reservierten Schlüsselwortes `synchronized` speziell gekennzeichnet wurden. Im Folgenden verwenden wir zum Senden einer Nachricht, die über eine Methode mit Synchronisationsmechanismus definiert ist, den Constraint `sendSynMsg` und zum Senden von Nachrichten an alle übrigen

Methoden wie bisher den Constraint `sendMsg`. Die zwei verschiedenen Arten von Nachrichten werden durch Einbettung in den folgenden abstrakten Datentyp unterschieden:

```
data Message a = MSG a | SYN_MSG a SynchroResult
```

In Java bewirkt die Kennzeichnung einer Methode mit `synchronized`, dass jeweils nur ein Prozess in diese Methode einsteigen kann. Auf diese Weise lassen sich kritische Bereiche absichern. In ObjectCurry hingegen kann es wie in jeder deklarativen Programmiersprache ohne Seiteneffekte keine kritischen Bereiche geben.

Wird eine Methode aufgerufen, die die Anweisung `wait r` enthält, so wird der aufrufende Prozess suspendiert und hinten an die Warteschlange `r` angehängt. Eine Methode mit der Anweisung `continue r` weckt den ersten Prozess aus der Warteschlange `r` auf und veranlasst ihn, seine zuletzt gesendete Nachricht zu wiederholen. Die Reihenfolge, in der Zuweisungen, Constraints und Anweisungen auf der rechten Seite einer Methode auftreten, hat keinen Einfluss auf die Semantik der Methode.

Betrachten wir nun die Benutzung des oben definierten Puffers. Auf einem Computer namens `agamemnon` wird ein Server durch Aufruf der Funktion `server` gestartet, der ein Puffer-Objekt erzeugt und über den Namen `"buffer"` veröffentlicht. Unabhängig davon in welcher Reihenfolge anschließend die Funktionen `client1` und `client2` auf zwei weiteren Rechnern gestartet werden, erhalten wir stets `"Received Values: 1 and 2"` als Ausgabe von `client2`.

```
server = newNamedObject "buffer" (buffer 1)

client1 = connectObject "buffer@agamemnon" >>= h
  h port | sendSynMsg (Put 2) port &> sendSynMsg (Put 3) port = done

client2 = connectObject "buffer@agamemnon" >>= h
  h port | sendSynMsg (Get x) port &> sendSynMsg (Get y) port
         = putStr "Received Values: "++(show x)++" and "++(show y)
```

4.2.2. Implementierung

Kommen wir nun zu der Implementierung des oben vorgestellten Monitorkonzepts.

Synchronisierte Methoden bestehen in ObjectCurry in der Regel aus zwei Teilen: Ein Teil behandelt den Fall, dass die Nachricht erfolgreich verarbeitet werden kann. Der andere Teil ist für das weitere Vorgehen zuständig, wenn der interne Zustand des Objekts die Verarbeitung der Nachricht nicht gestattet. Als Beispiel dient uns die Methode `Put` des auf Seite 42 vorgestellten Puffers: Ist der Puffer leer, so kann die Nachricht `Put` verarbeitet werden (`x := Just a`). Enthält der Puffer hingegen bereits ein Element, so ist dies nicht möglich. Die Aktion des Puffers muss daher in diesem Fall darin bestehen, den Sender der Nachricht zu suspendieren. Allgemein gilt: Wird der Teil einer Methode ausgeführt, der die Anweisung `delay r` enthält, so wird der Sender der Nachricht, die zum Aufruf dieses Teils der Methode führte, suspendiert. Wird hingegen ein anderer Teil ausgeführt, der keine Anweisung der Form `delay r` enthält, darf der Sender nicht suspendiert werden.

Realisieren lässt sich dieser Mechanismus durch einen Trick: Wir implementieren die Funktion `sendSynMsg` so, dass diese mit jeder Nachricht eine freie Variable c mit an das Objekt sendet. Anschließend überprüfen wir die Bindung dieser Variablen. Ein Teil einer Methode, der keine delay-Anweisung enthält, bindet c an den vordefinierten Wert `Ok`. Der Teil jedoch, der eine Anweisung der Form `delay r` enthält, bindet c nicht. Dadurch wird der sendende Prozess suspendiert, da er auf die Bindung von c warten muss. Im Folgenden nennen wir c **Synchronisationsvariable**. Sie besitzt den Typ `SynchroVar`:

```
data SynchroVar = Ok | Retry
```

Eine Synchronisationsvariable einer Nachricht erhält den Wert `Ok`, wenn die Nachricht erfolgreich verarbeitet wurde. Ansonsten ist sie ungebunden. Den Zweck von `Retry` werden wir weiter unten betrachten.

`sendSynMsg` wird so implementiert, wie wir es gerade beschrieben haben:

```
sendSynMsg m b =
    send (SYN_MSG m c) b & if (c==Ok) then success else sendSynMsg m b
      where c free
```

Damit ein auf diese Weise suspendierter Prozess zu einem späteren Zeitpunkt wieder aufgeweckt werden kann, legt die Methode mit der Anweisung `delay r` die nicht gebundene Synchronisationsvariable in die Warteschlange `r`. Eine Wartschlange besitzt den Typ `Queue`:

```
type Queue = [SynchroResult]
emptyQueue :: Queue
emptyQueue = []
```

Die Anweisung `continue r` dient zum Aufwecken eines suspendierten Prozesses. Das Aufwecken eines Prozesses ist sehr einfach: Es braucht nur eine der Synchronisationsvariablen aus `r` an einen beliebigen Wert gebunden werden, denn ein suspendierter Prozess wartet auf die Bindung dieser Variable. Allerdings ist die Nachricht, die zur Suspendierung des jetzt aufgeweckten Prozesses führte, noch nicht verarbeitet worden. Daher binden wir die Synchronisationsvariable an den speziellen Wert `Retry`. Dieser bewirkt, dass der aufgeweckte Prozess seine zuletzt gesendete Nachricht wiederholt, bevor er mit seiner eigentlichen Berechnung fortfährt. Realisiert ist dies durch den `else`-Zweig der Bedingung in der Funktion `sendSynMsg`, auf dessen Funktion wir bisher noch nicht eingegangen sind. Betrachten wir zum Abschluss die Übersetzung des oben vorgestellten Templates in Curry. Ein Puffer-Objekt versteht die Nachrichten `Put a`, `Get a` und `Stop`:

```
data Msg = Put Int | Get Int | Stop
```

Im Startzustand enthält das Attribut `x` des Puffers den Wert `Nothing` und `r1` sowie `r2` eine leere Warteschlange.

```
bufferInitState (i) = (Just i,emptyQueue,emptyQueue)
```

Die Konstruktorfunktion besteht aus dem üblichen Aufruf der Schleife `loop` mit der zugehörigen Action-Funktion:

```
buffer eval rigid
buffer i self = loop (bufferInitState (i)) bufferAction self
```

Innerhalb der Action-Funktion verwenden wir zwei zusätzliche Hilfsfunktionen. Der Constraint `retry` bindet den ersten Eintrag einer Warteschlange an den Wert `Retry`, falls die Warteschlange nicht leer ist, und ist sonst immer erfüllt.

```
retry :: Queue -> Success
retry l = if l==emptyQueue then success else (head l) =:= Retry
```

Die Funktion `rest` erwartet ebenfalls eine Warteschlange als Argument. Ist diese nicht leer, entfernt sie das erste Elemente und liefert die Restliste zurück; ist die Warteschlange leer, liefert sie die leere Warteschlange zurück.

```
rest :: Queue -> Queue
rest l = if l==emptyQueue then emptyQueue else tail l
```

In der folgenden Action-Funktion bezeichnet `synchroResult` die freie Variable, die über `sendSynMsg` mit an das Objekt geschickt wird (vgl. Zeilen 2, 8). Die Anweisung `delay r` führt innerhalb der Action-Funktion dazu, das die Variable `synchroResult` an die Warteschlange `r` angefügt wird (vgl. Zeilen 6, a). Enthält eine Bedingung der Methode keine Anweisung `delay r`, so wird `synchroResult` an `Ok` gebunden (Zeilen 3, b). Enthält eine Bedingung die Anweisung `continue r`, so wird das erste Element der Warteschlange `r` durch einen Aufruf von `retry r` an `Retry` gebunden (Zeilen 3, b). Außerdem wird dieses Element aus der Warteschlange entfernt (Zeilen 4, c).

```
0:  bufferAction eval rigid
1:  bufferAction (x,r1,r2) self (MSG Stop) = Finale
2:  bufferAction (x,r1,r2) self (SYN_MSG (Put a) synchroResult)
3:    | x =:= Nothing & (synchroResult =:= Ok) & retry r2    -- continue r2
4:      = State (Just a,r1,rest r2)
5:    | x =:= Just x2
6:      = State (x,r1 ++ [synchroResult],r2)                    -- delay r1
7:    where x2 free
8:  bufferAction (x,r1,r2) self (SYN_MSG (Get a) synchroResult)
9:    | x =:= Nothing
a:      = State (x,r1,r2 ++ [synchroResult])                    -- delay r2
b:    | x =:= Just x2 & a =:= x2 & (synchroResult =:= Ok) & retry r1
c:      = State (Nothing,rest r1,r2)                         -- continue r1
d:    where x2 free
```

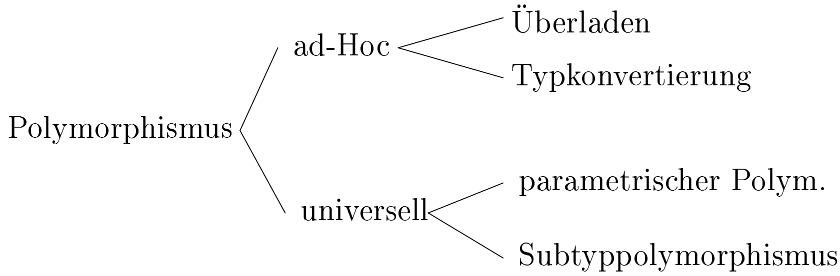
5. Erweiterungen des Typsystems

Objektorientierte Programmiersprachen erreichen ihre Flexibilität durch den Einsatz von Subtyppolymorphismus. Die Typsysteme funktionaler Programmiersprachen wie ML oder Haskell und auch das von Curry hingegen basieren auf parametrischem Polymorphismus. Beide Konzepte stehen orthogonal zueinander, d. h. keines von beiden kann durch das jeweils andere simuliert werden. Die Zielsetzung in der Entwicklung von ObjectCurry, nämlich die Implementierung möglichst vieler Merkmale konventioneller objektorientierter Programmiersprachen, macht es daher erforderlich, das Typsystem von Curry um Subtyppolymorphismus zu erweitern. Die Erweiterung von Hindley-Milner-Typsystemen um Subtyppolymorphismus ist ein viel beschriebenes Gebiet in der Literatur, wie die Arbeiten [Kae92], [Nor99b] zeigen. Die dort vorgestellten Konzepte gehen jedoch über unsere Erfordernisse hinaus, da in ObjectCurry Subtyppolymorphismus nur über Objekten und Nachrichten, nicht aber über den Standarddatentypen wie `Int` und `Real`, abstrakten Datentypen oder funktionalen Typen benötigt wird. Typsysteme wie das von Haskell oder Curry ermöglichen nicht nur die Überprüfung der Typen aller in einem Programm auftretenden Ausdrücke, sondern in vielen Fällen auch die automatische Inferenz dieser Typen.

Das folgende Kapitel geht zunächst kurz auf das Typsystem von Curry ein. Im Anschluss daran wird der Begriff der eingschränkten Typen eingeführt, um Subtyppolymorphismus auf Objekten und Nachrichten zu ermöglichen. Nach Vorstellung einer Core-Sprache, in die sich alle ObjectCurry-Programme übersetzen lassen, geben wir eine formale Definition eines Typsystems für ObjectCurry in Form von Inferenzregeln an. Dieses Typsystem dient als Grundlage eines Typinferenzalgorithmus, der in Kapitel 5.6 präsentiert wird.

5.1. Polymorphe Typsysteme

Bei Curry handelt es sich um eine streng getypte Programmiersprache, d. h. jeder Ausdruck besitzt einen eindeutigen Typ, der bereits zur Compilezeit bekannt ist. Wie Haskell verfügt Curry über ein Hindley-Milner-ähnliches polymorphes Typsystem [DM82]. Polymorph bedeutet, dass Ausdrücke oder Funktionen mehrere Typen besitzen können. Eine Reihe von funktionalen Programmiersprachen wie ML und Haskell basieren auf dem Konzept des **parametrischen Polymorphismus**, welches wir auch in Curry wiederfinden. Eine Klassifizierung der verschiedenen Arten von Polymorphismus nehmen Cardelli und Wegner [CW93] wie folgt vor:



Cardelli und Wegner unterscheiden zunächst zwischen ad-Hoc- und universellem Polymorphismus. **Ad-Hoc-Polymorphismus** findet sich in nahezu jeder getypten Programmiersprache: Der Operator `+` innerhalb von mathematischen Ausdrücken kann in der Regel für die Addition sowohl von Integer- bzw. Realzahlen als auch einer Mischung verwendet werden. Oft wird dies durch **Typkonvertierung** erreicht, d. h. bei der Addition `3 + 2.5` wird die Intger-Zahl `3` in eine Realzahl `3.0` umgewandet. Eine andere Art des ad-Hoc-Polymorphismus ist das **Überladen**. Überladen ermöglicht es, zwei oder mehr unterschiedlichen Funktionen innerhalb eines Sichtbarkeitsbereichs denselben Namen zu geben, falls ihre Typen sich unterscheiden. Über den Mechanismus des Überladens von Funktionen verfügen Programmiersprachen aus vielen Richtungen, so z. B. C++ [vG93], Java [CW97], Ada [Nag88], ObjectPascal 5 [Inp96] oder auch Haskell[1].

Interessanter sind die Formen des **universellen Polymorphismus**. **Subtyppolymorphismus** findet sich in nahezu jeder objektorientierten Programmiersprache. Er ermöglicht an jeder Stelle, an der das Typsystem ein Objekt einer bestimmten Klasse C erwartet, auch die Verwendung von Objekten einer Unterklasse zu C. Zusammen mit den Mechanismen der Vererbung (s. Kapitel 1.1.1) und des dynamischen Bindens (s. Kapitel 3.2.3) erreichen objektorientierte Programmiersprachen eine höhere Flexibilität als klassische, strukturierte Programmiersprachen wie Pascal.

Parametrisch polymorphe Typen enthalten eine oder mehrere **Typvariablen**. Diese können durch beliebige andere Typen ersetzt werden[2]. Auf diese Weise entsteht ein konkreter Typ, der **Instanz** genannt wird. Wie bereits angesprochen basieren die Typsysteme von ML, Haskell und Curry auf parametrischem Polymorphismus. Aber auch C++ und Ada kennen eine Art parametrischen Polymorphismus, indem sie die Definition von Schablonen erlauben, die vom Compiler mit unterschiedlichen Typen ausgefüllt werden.

5.2. Das Typsystem von Curry

Curry gestattet wie Haskell parametrischen Polymorphismus. Datenkonstruktoren müssen explizit getypt werden, d. h. ihr Typ muss im Programm angegeben sein. Der Typ von Funktionen kann explizit angegeben werden, er darf aber auch fehlen. In diesem Fall versucht der Compiler mittels Typinferenz den Typ der Funktion zu ermitteln.

[1] Das Konzept der Typklassen dient dem Überladen von Funktionen.

[2] In einigen Sprachen ist es möglich, nicht beliebige Typen zuzulassen, sondern bestimmte Einschränkungen zu treffen, so ist z. B. in Haskell die Angabe von Typklassen möglich, zu denen ein Typ gehören muss.

Wir geben zunächst eine formale Definition des Typsystem von Curry an, wie sie auch in [Han00a] zu finden ist.

Typausdrücke sind in Curry wie folgt definiert:

Def. 5.2.1 (Typausdrücke) *Ein **Typausdruck** ist eine Typvariable $\alpha \in \mathcal{V}$, ein Basistyp wie* `Bool`, `Success`, `Int`, `Float`, `Char` *oder die Applikation eines n-stelligen Typkonstruktors $T \in F$ auf die Typen $\tau_1 \dots \tau_n$ der Form $(T\ \tau_1\ \dots\ \tau_n)$, wobei $\tau_1, \dots, \tau_n$ Typausdrücke sind.*

Die Menge der Typvariablen bezeichnen wir mit $\mathcal{V}$ und die Menge der Typkonstruktoren mit F. Typen, dargestellt durch die entsprechenden Typausdrücke, sind Elemente der Termalgebra $T_F(\mathcal{V})$. Funktionale Typausdrücke werden durch eine spezielle Schreibweise angegeben: Wir schreiben $a \rightarrow b$ für den Typ einer Funktion mit dem Definitionsbereich a und dem Wertebereich b.

Typausdrücke, die Typvariablen beinhalten, heißen polymorph, während Typausdrücke ohne Typvariablen monomorph genannt werden. Gleiche Variablen innerhalb eines Typausdrucks drücken Abhängigkeiten zwischen Typen aus [Car87]. So hat z. B. die Identitätsfunktion den Typ $a \rightarrow a$, was bedeutet, dass Definitions- und Wertebereich dieser Funktion gleich sind. Von polymorphen Typen können Instanzen durch Ersetzen der Typvariablen mit anderen Typausdrücke gebildet werden. So ist z. B. `Bool` $\rightarrow$ `Bool` eine Instanz von $a \rightarrow a$. Formal definieren wir hierzu den Begriff der **Substitution** (vgl. [Kae92]).

Def. 5.2.2 (Substitution)
*Eine **Substitution** σ ist eine Abbildung von Typvariablen auf Typausdrücke, d. h. $\sigma \in \mathcal{V} \rightarrow T_F(\mathcal{V})$, mit $\sigma(\alpha) \neq \alpha$ für nur endlich viele $\alpha \in \mathcal{V}$. Schreibweise: $\sigma = [x_1/\tau_1, \dots, x_n/\tau_n]$ falls $\sigma(x_i) = \tau_i$ für alle $i = 1, \dots, n$ und $\sigma(y) = y$ für alle $y \notin \{x_1, \dots, x_n\}$. Eine Substitution lässt sich kanonisch auf Typen zu einer Funktion $\sigma \in T_F(\mathcal{V}) \rightarrow T_F(\mathcal{V})$ erweitern.*

Def. 5.2.3 (Instanz)
Ein Typ τ_2 ist eine Instanz eines Typs τ_1, falls eine Substitution σ existiert, so dass $\tau_2 = \sigma\tau_1$.

Zur Definition des Typsystems ist es notwendig, innerhalb von Typausdrücken zwischen solchen Typvariablen zu unterscheiden, die vom jeweiligen Kontext abhängig sind (**spezifische Typvariablen**), und solchen, die frei belegt werden können (**generische Typvariablen**).

Def. 5.2.4 (Typschema)
*Ein **Typschema** ist ein Typ, verbunden mit einer Liste, die die generischen Typvariablen dieses Typs enthält. Schreibweise: $\forall\alpha_1, \dots, \alpha_n.\tau$, wobei τ ein Typ und $\alpha_1, \dots, \alpha_n$ die darin enthaltenen generischen Typvariablen sind.*

Def. 5.2.5 (generische Instanz)
*Ein Typ τ' heißt **generische Instanz** eines Typschems $\forall\alpha_1, \dots, \alpha_n.\tau$, falls eine Substitution σ existiert, so dass $\tau' = \sigma\tau$ mit $\sigma(\alpha_i) = \tau_i$ für alle $i = 1, \dots, n$, $\sigma(\beta) = \beta$ für alle $\beta \notin \{\alpha_1, \dots, \alpha_n\}$.*

Jede Funktion eines Curryprogramms besitzt einen Typ. Da diese Typen generische Typvariablen enthalten können, wird jedem Funktionsbezeichner statt eines Typs ein Typschema zugeordnet. Diese Zuordnung erfolgt durch eine Typumgebung:

Def. 5.2.6 (Typumgebung)
Eine ***Typumgebung*** Γ *ist eine Funktion, die einen Bezeichner auf ein Typschema abbildet.*

In der Typumgebung können wir die Typen der Funktionen nachschlagen. Ausgehend von einer Typumgebung, die alle in einer Funktionsdefinition auftretenden Bezeichner auf Typschemata abbildet, können wir die Wohlgetyptheit einer definierenden Gleichung definieren:

Def. 5.2.7 (Wohlgetyptheit einer definierenden Gleichung)
Eine definierende Gleichung $f\ t_1\ \ldots\ t_n$ `=` e *ist* ***wohlgetypt*** *bezüglich einer Typumgebung* Γ*, falls*

- $\Gamma(f) = \forall\alpha_1, \ldots, \alpha_n.\tau$ *und*
- $\Gamma \vdash \backslash t_1$ `->` $\cdots$ `->` $\backslash t_n$ `->` $e{:}\tau$ *mit den in Tabelle 5.1 abgebildeten Inferenzregeln herleitbar ist.*

Dabei ist es notwendig, sämtliche lokale Funktionsdeklarationen in einem Curry-Programm durch eine syntaktische Transformationen zu eleminieren. Ein solches Verfahren findet sich in [Han00a], Anhang D.8. Eine Gleichung der Form `f | c = e` wird als syntaktischer Zucker für `f = cond c e` gesehen, wobei `cond` den Typ `Success` `->` α `->` α besitzt.

Die Inferenzregeln aus Tabelle 5.1 erlauben die Herleitung des Typs eines Ausdrucks, falls eine Typumgebung Γ vorgegeben ist. Die Regel [Axiom] ist in folgender Weise zu lesen: Bei gegebener Typumgebung Γ besitzt der Ausdrucks x den Typ τ, falls τ eine generische Instanz von $\Gamma(x)$ ist. Die Regel schaut also den Typ der Variablen x in der Typumgebung Γ nach. Zur Anwendung aller übrigen Regeln müssen bestimmte Voraussetzungen erfüllt sein, die oberhalb der horizontalen Linie angegeben sind. Die Abstraktionsregel [Abstraktion] besagt, dass ein Ausdruck der Form $\backslash x$`->`e bei gegebener Typumgebung Γ den Typ $\tau \to \tau'$ besitzt. Dies gilt jedoch nur, falls bei Veränderung der Typumgebung Γ in der Weise, dass x das Typschema τ zugeordnet wird, e den Typ τ' besitzt. Die Regel [FreeVars] erweitert ebenfalls die Typumgebung Γ, indem sie einer Variablen x ein Typschema τ zuordnet. [Applikation] ordnet der Applikation eines Ausdrucks $e1$ auf einen Ausdruck $e2$ den Ergebnistyp von $e1$ zu, falls $e2$ denselben Typ wie das Argument von $e1$ besitzt. Die Regel [Bedingung] prüft, ob die Bedingung eines `if...then...else`-Ausdrucks ein boolscher Ausdruck ist, und ob then- und else-Zweig denselben Typ besitzen.

Wir benötigen keine Regel zur Behandlung des `let`, da sämtliche lokalen Deklarationen wie oben beschrieben elemeniert werden.

Def. 5.2.8 (Wohlgetyptheit eines Curry-Programms) *Ein Programm heißt* ***wohlgetypt****, falls alle seine Regeln wohlgetypt bezüglich einer einzigen Typumgebung sind.*

[Axiom]	$\dfrac{}{\Gamma \vdash x : \tau}$ falls τ generische Instanz von $\Gamma(x)$
[Abstraktion]	$\dfrac{\Gamma[x/\tau] \vdash e : \tau'}{\Gamma \vdash \backslash x \texttt{ -> } e : \tau \rightarrow \tau'}$ falls τ Typausdruck
[FreeVars]	$\dfrac{\Gamma[x/\tau] \vdash e : \tau'}{\Gamma \vdash \texttt{let}\ x\ \texttt{free in}\ e : \tau'}$ falls τ Typausdruck
[Applikation]	$\dfrac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$
[Bedingung]	$\dfrac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$

Tabelle 5.1.: Typregeln für Curry-Programme

5.3. Subtypen in ObjectCurry

Das Typsystem von ObjectCurry baut auf dem von Curry auf. Zusätzlich zu den vordefinierten Standarddatenkonstruktoren existieren drei einstellige Konstruktoren `Constructor`, `Message` und `Object` zur Darstellung der Typen von Konstruktoren, Nachrichten und Objekten. Dabei gibt der Parameter dieser Datenkonstruktoren an, zu welchem Template der Konstruktor, das Objekt bzw. die Nachricht gehört.

Zunächst geben wir an, welche Typen Konstruktoren, Objekte und Nachrichten sowie die Funktionen `send` und `new` besitzen, wenn wir von einem Hindley-Milner-ähnlichen Typsystem wie in Curry ausgehen.

Ein Template T führt, wie in Kapitel 3.1.1 vorgestellt wurde, die Bezeichner der Konstruktorfunktion und der einzelnen Nachrichten ein. Eine 0-stellige Konstruktorfunktion erhält den Typ

$$\texttt{Constructor}\ T$$

Erwartet die Konstruktorfunktion hingegen n Argumente mit den Typen $\tau_1, \ldots, \tau_n$, so geben wir ihr den Typ

$$\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \texttt{Constructor}\ T$$

Wir erlauben keine polymorphen Templates. Daher muss auch die Konstruktorfunktion monomorph sein: Die Typausdrücke $\tau_1, \ldots, \tau_n$ dürfen keine Typvariablen enthalten.

Den Typ einer Nachricht wählen wir vorerst analog, werden aber später eine neue Definition angeben, um Subtyppolymorphismus zu ermöglichen. Eine 0-stellige Nachricht, die durch eine Methode des Templates T definiert ist, erhält den Typ

$$\texttt{Message}\ T$$

Eine n-stellige Nachricht eines Templates T ist vom Typ

$$\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \texttt{Message}\ T$$

Auch hier sind $\tau_1, \ldots, \tau_n$ die Typen der Argumente. Eine redefinierte Methode führt keine neue Nachricht ein. Die Typen ihrer Argumente müssen daher mit denen der überschrieben Methode übereinstimmen.

Bsp. 5.3.1

Wir betrachten noch einmal die in Kapitel 3 vorgestellten Templates `Counter` *und* `MaxCounter`*. Die Konstruktorfunktionen* `counter` *und* `maxCounter` *sowie die Nachrichten* `Inc`*,* `Get`*,* `Set` *und* `SetMax` *erhalten die folgenden Typen:*

`counter`	`Int` $\rightarrow$ `Constructor Counter`
`maxCounter`	`Int` $\rightarrow$ `Int` $\rightarrow$ `Constructor MaxCounter`
`Inc`	`Message Counter`
`Get`	`Int` $\rightarrow$ `Message Counter`
`Set`	`Int` $\rightarrow$ `Message Counter`
`SetMax`	`Int` $\rightarrow$ `Message MaxCounter`

In einem ersten Ansatz geben wir der Funktion `send` zum Senden von Nachrichten den Typ

$$\texttt{Message}\ t \rightarrow \texttt{Object}\ t \rightarrow \texttt{Success}$$

Hierbei ist t eine freie, generische Typvariable, die den Typ des Templates angibt.

`new` erhält den Typ

$$\texttt{Constructor}\ t \rightarrow \texttt{Object}\ t \rightarrow \texttt{Success}$$

Durch Anwendung von `new` auf einen Konstruktor des Typs T, erhält das erzeugte Objekt den Typ

$$\texttt{Object T}$$

Alle bisher vorgestellten Typen fassen wir in Tabelle 5.2 noch einmal zusammen.

In einem Hindley-Milner-ähnlichen Typsystem wie dem von Curry erlauben die auf diese Weise definierten Typen von Konstruktoren, Nachrichten und Objekten keinen Subtyppolymorphismus. Insgesamt gibt es jedoch drei Situationen, in denen Subtyppolymorphismus gewünscht wird:

Konstruktor	$\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow$ `Object` T
Nachricht	$\tau_1 \rightarrow \cdots \rightarrow \tau_m \rightarrow$ `Message` T
Objekt	`Object` T
send	`Message` $t \rightarrow$ `Object` $t \rightarrow$ `Success`
new	`Constructor` $t \rightarrow$ `Object` $t \rightarrow$ `Success`

Tabelle 5.2.: Typen in ObjectCurry ohne Subtyppolymorphismus

1. Wir möchten einem Objekt eines Templates t nicht nur Nachrichten schicken können, die in t definiert werden, sondern auch Nachrichten aus den Vorfahren von t.
2. Wir möchten Objekte verschiedener Templates in einer gemeinsamen Datenstruktur, z. B. einer Liste, ablegen können, wenn Templates dieser Objekte einen gemeinsamen Vorfahren besitzen. Alle Objekte in der Datenstruktur verstehen dann nämlich die Nachrichten dieses gemeinsamen Vorfahren.
3. Auch Nachrichten verschiedener Templates möchten wir in einer gemeinsamen Datenstruktur, z. B. einer Liste, ablegen können, wenn die Templates, in denen diese Nachrichten definiert sind, ein gemeinsames Subtemplate besitzen. Da wir keine Mehrfachvererbung zulassen, liegen solche Templates in einer Kette. Wir können einem Objekt des gemeinsamen Subtemplates alle Nachrichten der Datenstruktur schicken.

Die bisher gewählten Typen erlauben keinen Subtyppolymorphismus. So ist es beispielsweise nicht möglich, an ein Objekt eine Nachricht zu schicken, die in einem Vorfahren definiert wurde:

Bsp. 5.3.2
Sei `maxCounterObject` *ein Objekt vom Typ* `Object MaxCounter`*. Der Ausdruck*

```
send Inc maxCounterObject
```

ist nicht wohlgetypt, da `Inc` *den Typ* `Message Counter` *besitzt.*

Wählen wir für `send` den Typ

$$\texttt{Message } t_1 \to \texttt{Object } t_2 \to \texttt{Success}$$

so ist dieser Typ zu allgemein, denn nun können alle Nachrichten an ein Objekt geschickt werden, also auch solche, die das Objekt nicht versteht. Wir benötigen eine Möglichkeit, diesen Typ so einzuschränken, dass nur Nachrichten an eine Instanz des Templates `t2` geschickt werden können, die innerhalb von `t2` oder eines Vorfahren definiert sind. Zu diesem Zweck führen wir Typconstraints über Templates ein.

Def. 5.3.3 (Typconstraints)
Die Menge der ***Typconstraints*** *über $T_F(V)$ ist definiert als $\{p(\tau_1, \ldots, \tau_n) | p \in P_n, \tau_i \in T_F(V)\}$, falls P eine endliche n-indizierte Familie von Prädikatsymbolen ist.*

Für unsere Zwecke genügt es, Subtypconstraints über Templates zu betrachten. Dazu wählen wir

$$P = P_2 = \{\leq\}$$

und beschränken F auf die Namen der in einem Programm definierten Templates. Wir schreiben einen Subtypconstraint in der Form $\tau_1 \leq \tau_2$.

Analog zu [Kae92] und [Nor99a] definieren wir nun den Begriff des eingeschränkten Typs:

Def. 5.3.4 (Eingeschränkte Typen)
Ein ***eingeschränkter Typ*** *ist ein Paar $\tau | C$, bestehend aus einem Typausdruck τ und einer Menge von Typconstraints C [Kae92]. Anstatt $\tau | \emptyset$ schreiben wir auch τ.*

Mit diesen Hilfsmitteln lässt sich nun auch der gewünschte Typ von `send` angeben:

$$\texttt{Message } t_1 \rightarrow \texttt{Object } t_2 \rightarrow \texttt{Success | } \{t_2 \leq t_1\}$$

Über den Constraint $t_2 \leq t_1$ drücken wir aus, dass das Template t_2 ein Subtemplate zu t_1 sein muss. Auf diese Weise ist die Benutzung von `send` so eingeschränkt, dass eine Nachricht, die im Template t_1 definiert wurde, nur dann an eine Instanz von t_2 geschickt werden kann, wenn t_1 ein Vorfahre von t_2 ist. Dabei muss t_1 kein direkter Vorfahre von t_2 sein.

Bsp. 5.3.5
Sei `maxCounterObject` *ein Objekt vom Typ* `Object MaxCounter`*. Der Ausdruck*

```
send Inc maxCounterObject
```

besitzt nun den Typ

$$\texttt{Success | } \{\texttt{MaxCounter } \leq \texttt{Counter}\}$$

Wir sagen, der Constraint `MaxCounter` $\leq$ `Counter` *ist erfüllt, denn die* `extends`*-Klausel des Templates* `MaxCounter` *führt dazu, dass* `Counter` *ein Vorfahre von* `MaxCounter` *ist.*

Es hängt vom jeweiligen Programm ab, ob ein Constraint erfüllt ist oder nicht. Genauer gesagt ist es entscheidend, in welcher Beziehung die in einem Programm definierten Templates zueinander stehen. Wir möchten als nächstes die Erfüllbarkeit von Constraints formal definieren. Dabei gibt uns die Vererbungshierachie Aufschluss über die Beziehungen zwischen den Templates eines Programms:

Def. 5.3.6 (Vererbungshierarchie)
Sei $\mathcal{H}$ die durch ein Programm P definierte Relation der direkten Vorfahren. Dann heisst die zugehörige reflexive und transitive Hülle $\mathcal{H}_P^*$ ***Vererbungshierachie****. Wir schreiben im Folgenden $\mathcal{H}^*$ anstatt $\mathcal{H}_P^*$, da wir uns immer auf ein bestimmtes Programm beziehen* *(s. Kapitel 3.2, S. 3.2).*

Def. 5.3.7 (Reflexive und transitive Hülle)
Sie R eine zweistellige Relation über einer Menge M. Die transitive Hülle R^+ sowie die reflexive und transitive Hülle R^* von R sind wie folgt definiert:

$$\begin{aligned} R^* &:= \textstyle\bigcup_{n=0}^{\infty} R^n \\ R^+ &:= \textstyle\bigcup_{n=1}^{\infty} R^n \\ R^0 &:= \{(m,m)|m \in M\} \\ R^{n+1} &:= R \circ R^n \end{aligned}$$

Dabei ist das Relationenprodukt $R_1 \circ R_2$ zweier Relationen definiert durch:

$$(m_1, m_2) \in (R_1 \circ R_2) :\Leftrightarrow \exists m \in M : (m_1, m) \in R_1 \text{ und } (m, m2) \in R_2$$

Def. 5.3.8 (Erfülltheit von Subtypconstraints)
Eine Substitution σ erfüllt einen Subtypconstraint $\tau_1 \leq \tau_2$ bzgl. der Vererbungshierachie $\mathcal{H}^*$ (Schreibweise: $\sigma \models_{\mathcal{H}^*} \tau_1 \leq \tau_2$), falls eine Substitution σ existiert, so dass $(\sigma\tau_1, \sigma\tau_2) \in \mathcal{H}^*$.

Eine Substitution σ erfüllt eine Menge von Subtypconstraints C (Schreibweise: $\sigma \models_{\mathcal{H}^*} C$), falls für alle $c \in C$ gilt: $\sigma \models_{\mathcal{H}^*} c$.

Eine Menge von Subtypconstraints C ist erfüllbar bzgl. der Vererbungshierachie $\mathcal{H}^*$ (Schreibweise: $\models_{\mathcal{H}^*} C$), falls eine Substitution σ existiert, so dass gilt: $\sigma \models_{\mathcal{H}^*} C$.

Bsp. 5.3.9
Sei `counterObject` *eine Instanz des Templates* `Counter` *und* `maxCounterObject` *eine Instanz von* `MaxCounter`. *Ferner gehen wir von den folgenden Typannahmen aus:*

```
Inc              :: Message Counter
SetMax           :: Int → Message MaxCounter
counterObject    :: Object Counter
maxCounterObject :: Object MaxCounter
```

Die folgenden Ausdrücke besitzen dann die jeweils angegebenen Typen:

```
send Inc counterObject              :: Success | {Counter ≤ Counter}
send (SetMax 42) maxCounterObject   :: Success | {MaxCounter ≤ MaxCounter}
send Inc maxCounterObject           :: Success | {MaxCounter ≤ Counter}
send (SetMax 42) counterObject      :: Success | {Counter ≤ MaxCounter}
```

Bezüglich der Vererbungshierachie $\{(\texttt{MaxCounter}, \texttt{Counter})\}^*$ *sind die Constraints der Typen der ersten drei Ausdrücke erfüllt, nicht jedoch der Constraint in der letzten Zeile, da* `MaxCounter` *kein Vorfahre von* `Counter` *ist.*

Wir haben gesehen, dass wir mit Hilfe von Subtypconstraints einen Typen von `send` angeben können, der es uns ermöglicht, Nachrichten eines Templates auch an Instanzen eines Subtemplates zu schicken. Die Funktion `send` ist polymorph, und zwar handelt es sich bei dieser Art von Polymorphismus um Subtyppolymorphismus, wie wir ihn auch aus konventionellen objektorientierten Programmiersprachen kennen.

Auch bei der `new`-Funktion werden wir nun Subtyppolymorphismus zulassen. Bisher sind wir davon ausgegangen, dass Instanzen eines Templates T den Typ `Object` T besitzen.

Um Objekte verschiedener Typen in einer Datenstruktur ablegen zu können, verändern wir den Typ der `new`-Funktion:

$$\texttt{Constructor}\ t_1 \rightarrow \texttt{Object}\ t_2 \rightarrow \texttt{Success}\ |\ \{t_1 \leq t_2\}$$

Bsp. 5.3.10
Der Constraint `new (counter 0) o1` *erzeugt ein Objekt* `o1` *des Typs*

$$\texttt{Object t}\ |\ \{\texttt{Counter} \leq t\}$$

`new (maxCounter 0 42) o2` *erzeugt ein Objekt* `o2` *des Typs*

$$\texttt{Object t}\ |\ \{\texttt{MaxCounter} \leq t\}$$

Bilden wir nun die Liste `[o1,o2]`, *so besitzt diese den Typ*

$$\texttt{List (Object}\ t\texttt{)}\ |\ \{\texttt{Counter} \leq t, \texttt{MaxCounter} \leq t\}$$

Die Menge der Constraints dieses Typs ist bezüglich der Vererbungshierachie

$$\{(\texttt{MaxCounter}, \texttt{Counter})\}^*$$

erfüllbar (Substitution: $\sigma(t) = \texttt{Counter}$).

Diese Definition der `new`-Funktion ermöglicht also Subtyppolymorphimus über Objekten. Zuletzt möchten wir noch Subtyppolymorphismus über Nachrichten zulassen, um beispielsweise eine Liste bilden zu können, die Nachrichten verschiedener Typen enthalten kann. Dazu legen wir fest, dass eine n-stellige Nachricht M, die in einem Template T definiert wird, folgenden Typ besitzt:

$$\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \texttt{Message}\, t \;\texttt{|}\; \{t \leq T\}$$

Bsp. 5.3.11
Diese Festlegung ermöglicht die Bildung einer Liste mit den Elementen `Inc` *und* `(SetMax 42)`*:* `[Inc,(SetMax 42)]` *besitzt den Typ*

$$\texttt{List (Message } t\texttt{)} \;\texttt{|}\; \{t \leq \texttt{Counter}, t \leq \texttt{MaxCounter}\}$$

Das folgende Beispiel zeigt, dass auf die Constraints der `send`-Funktion nicht verzichtet werden kann.

Bsp. 5.3.12
Wir betrachten die folgende Definition der Funktion `f`*:*

```
f m1 m2 o1 o2 = send m1 o1 & send m2 o2 & send m1 o2
```

`f` *nimmt zwei Nachrichten und zwei Objekte als Argumente und verschickt die erste Nachricht an das erste Objekt, die zweite Nachricht an das zweite Objekt und zusätzlich auch die erste Nachricht an das zweite Objekt.*

Gehen wir von einer `send`*-Funktion ohne Subtyppolymorphismus aus, so besäße* `f` *den folgenden Typ:*

$$\texttt{Message } a \rightarrow \texttt{Message } a \rightarrow \texttt{Object } a \rightarrow \texttt{Object } a \rightarrow \texttt{Success}$$

Weiterhin gehen wir von den folgenden Typannahmen aus:

$$\begin{array}{lll}
\texttt{Inc} & :: & \texttt{Message } a \;\texttt{|}\; \{a \leq \texttt{Counter}\} \\
\texttt{(SetMax 42)} & :: & \texttt{Message } a \;\texttt{|}\; \{a \leq \texttt{MaxCounter}\} \\
\texttt{counterObject} & :: & \texttt{Object } a \;\texttt{|}\; \{\texttt{Counter} \leq a\} \\
\texttt{maxCounterObject} & :: & \texttt{Object } a \;\texttt{|}\; \{\texttt{MaxCounter} \leq a\}
\end{array}$$

Dann besitzt der Ausdruck

```
f Inc (SetMax 42) counterObject maxCounterObject
```

den Typ

$$\texttt{Success | } \{a \leq \texttt{Counter}, a \leq \texttt{MaxCounter}, \texttt{Counter} \leq a, \texttt{MaxCounter} \leq a\}$$

Die angegebenen Constraints sind nicht erfüllbar, da keine Substitution für `a` *existiert, so dass alle Constraints Elemente der Vererbungshierachie sind. Dies widerspricht unseren Erwartungen, denn es ist durchaus möglich,* `Inc` *an* `Counter` *und* `MaxCounter` *sowie* `SetMax 42` *an* `MaxCounter` *zu senden.*

Nun verwenden wir `send` *mit Subtyppolymorphismus:*

$$\texttt{Message } t_1 \to \texttt{Object } t_2 \to \texttt{Success | } \{t_2 \leq t_1\}$$

Als Typ für `f Inc (SetMax 42) counterObject maxCounterObject` *erhalten wir nun:*

$$\texttt{Message } a \to \texttt{Message } b \to \texttt{Object } c \to \texttt{Object } d \to \texttt{Success | } \{c \leq a, d \leq b, d \leq a\}$$

Wenden wir `f` *nun auf dieselben Ausdrücke wie oben an, so erhalten wir*

$$\begin{array}{l}\texttt{Success |}\\ \{c \leq a, d \leq b, d \leq a, a \leq \texttt{Counter}, b \leq \texttt{MaxCounter}, \texttt{Counter} \leq c, \texttt{MaxCounter} \leq d\}\end{array}$$

Die Constraints dieser Gleichung sind bezüglich der Vererbungshierachie $\{(\texttt{MaxCounter}, \texttt{Counter})\}^*$ *durch die folgende Substitution* σ *lösbar:*

$$\begin{aligned}\sigma(a) &= \texttt{Counter}\\ \sigma(b) &= \texttt{MaxCounter}\\ \sigma(c) &= \texttt{Counter}\\ \sigma(d) &= \texttt{MaxCounter}\end{aligned}$$

Als nächstes möchten wir das von uns verwendete Typsystem mit Typconstraints formal definieren. Zur Vereinfachung tun wir dies jedoch nicht über die gesamte Sprache ObjectCurry, sondern nur über eine Core-Sprache, in die sich jedes ObjectCurry-Programm transformieren lässt. Der folgende Abschnitt stellt diese Core-Sprache vor.

5.4. Core ObjectCurry

Im Folgenden wird eine **Core-Sprache** für ObjectCurry vorgestellt. Jedes ObjectCurry-Programm lässt sich in ein äquivalentes Programm dieser Core-Sprache übersetzen. Dabei gehen wir davon aus, dass lokale Deklarationen zuvor mittels des in [Han00a], Anhang D.8 beschriebenen Verfahrens eleminiert wurden.

Def. 5.4.1 *Ausdrücke und Templates der Core-Sprache sind durch die EBNF-Regeln aus Tabelle 5.3 definiert.*

Ein Ausdruck E der Core-Sprache ist wie in Curry eine Variable, eine Lambda-Abstraktion, eine Applikation zweier Ausdrücke, ein Ausdruck mit der Deklaration einer freien Variable oder ein bedingter Ausdruck der Form `if...then...else`. Ein Template T besteht aus den initialen Zuweisungen `I`, die die Attribute und Initialwerte festlegen, sowie einer beliebigen Anzahl von Methoden `M`*. Ein Template kann durch die Angabe `extends` als

E	$::=$	x	Variable
	$\mid$	$\lambda x.E$	Abstraktion über Ausdruck
	$\mid$	$E_1\ E_2$	Applikation
	$\mid$	$\texttt{let}\ x\ \texttt{free in}\ E$	freie Variable
	$\mid$	$\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3$	Bedingung
T	$::=$	$\texttt{Template}\ name\ I\ M^*$	Template
	$\mid$	$\texttt{Template}\ name_1$ $\texttt{extends}\ name_2\ I'\ M^*$	Subtemplate
A	$::=$	$(x := E)^*$	Zuweisungen
I	$::=$	A	initiale Zuweisung
	$\mid$	$\lambda x.I$	Abstraktion über initialen Zuweisungen
I'	$::=$	E, A	initiale Zuweisungen eines Subtemplates
	$\mid$	$\lambda x.I'$	Abstr. über init. Zuw. eines Subtemplates
M	$::=$	$E \Rightarrow A$	Methodenrumpf
	$\mid$	$\lambda x.M$	Abstraktion über Methodenrumpf

Tabelle 5.3.: Eine Core-Sprache für ObjectCurry

Subtemplate eines anderen Templates definiert werden. I' enthält zusäztlich zu den initialen Zuweisungen der in diesem Subtemplate neu definierten Attribute den Aufruf der Konstruktorfunktion des Vorfahren. Dadurch erhält auch jedes geerbte Attribut einen Initialwert.

Ein Core ObjectCurry-Programm ist eine Menge von Funktions- und Templatedefinitionen. Eine Funktionsdefinition hat die Form

$$functionName = E,$$

eine Templatedefinition die Form

$$(constructorName, methodeName_1, \ldots, methodName_n) = T.$$

Ein Core ObjectCurry-Programm enthält keine lokalen Definitionen: Alle Bezeichner werden auf Top-Level eingeführt. Man erkennt, dass der Name einer Funktion außerhalb des Ausdrucks E festgelegt wird, der diese beschreibt. Analog werden auch die Namen von Konstruktoren und Methoden außerhalb von T angegeben.

Ein Block von Zuweisungen A besteht aus einer beliebigen Anzahl von Zuweisungen der Form $x := E$, wobei E ein beliebiger Ausdruck ist. Ein Block initialer Zuweisungen eines Templates I ist im Wesentlichen ein Block aus Zuweisungen. Dieser bestimmt die Attribute eines Templates und deren Initialwerte. Er übernimmt damit die Aufgabe der Konstruktorfunktion eines Templates aus ObjectCurry. Eine Konstruktorfunktion in ObjectCurry kann jedoch auch eine mehrstellige Funktion sein, die zusätzliche Parameter erwartet. Aus diesem Grund lassen wir Lambda-Abstraktion über initialen Zuweisungen zu. Gleiches gilt für die initialen Zuweisungen eines Subtemplates.

Eine Methode M ist definiert durch einen Ausdruck E und einen Block von Zuweisungen A. Der Ausdruck E gibt einen Constraint an, der beim Aufruf der Methode gelöst werden muss. ObjectCurry erlaubt die Angabe mehrerer Constraints innerhalb eines Methodenrumpfs; diese können jedoch durch Verwendung von & zu einem einzigen Constraint konkateniert werden. Wie in ObjectCurry beschreibt A die gewünschte Zustandstransformation. Auch über Methoden lassen wir Lambda-Abstraktion zu, da Methoden ebenfalls eine beliebige Anzahl von Parametern erwarten können.

Bsp. 5.4.2 *Als Beispielprogramm dient uns erneut der Zähler aus Kapitel 3.1.1. In die Core-Sprache übertragen, besitzt er folgendes Aussehen:*

$$
\begin{array}{rll}
(\texttt{counter}, \texttt{Inc}, \texttt{Get}) = & \texttt{Template Counter} & \\
& \lambda i \,.\, x := i & \textit{(Rumpf von } \texttt{counter}\textit{)} \\
& \texttt{success} \Rightarrow x := x + 1 & \textit{(Rumpf von } \texttt{Inc}\textit{)} \\
& \lambda v \,.\, (v =:= x) \Rightarrow \epsilon & \textit{(Rumpf von } \texttt{Get}\textit{)}
\end{array}
$$

$$
\begin{array}{rl}
(\texttt{maxCounter}, \texttt{Inc}, \texttt{SetMax}) = & \texttt{Template MaxCounter extends Counter} \\
& \lambda i \,.\, \lambda mi \,.\, counter\ i, max := mi \\
& \texttt{success} \Rightarrow x := \texttt{if}\ x \texttt{<} max\ \texttt{then}\ x + 1\ \texttt{else}\ x \\
& \lambda v \,.\, \texttt{success} \Rightarrow max := v
\end{array}
$$

Die Konstruktorfunktion zur Erzeugung eines Counter-Objekts ist über den Bezeichner `counter` zugänglich. Die Methoden erhalten die Namen `Inc` und `Get`. Da die Nachricht `Get` einen weiteren Parameter verlangt, wird die Definition über eine Lambda-Abstraktion vorgenommen.

`MaxCounter` erbt von `Counter`. Die geerbten Attribute erhalten durch den Aufruf der Konstruktorfunktion des Vorfahren ($counter\ i$) ihre Initialwerte. Die Methode `Inc` wird überschrieben und die Methode `SetMax` neu eingeführt.

Wir verwenden den vordefinierten, immer erfüllten Constraint `success` in den Methoden, die die Lösung keines weiteren Constraints erfordern.

5.5. Ein Typsystem für ObjectCurry

Zur Definition des Typsystems müssen die Begriffe des Typschemas und der Typumgebung um Constraints erweitert werden:

Def. 5.5.1 (Eingeschränktes Typschema) *Ein* ***eingeschränktes Typschema*** *ist ein Typschema der Form* $\forall \alpha_1, \ldots, \alpha_n.\tau | C$*, wobei* $\forall \alpha_1, \ldots, \alpha_n.\tau$ *ein Typschema und* C *eine Menge von Constraints ist.*

Die Variablen $\alpha_1, \ldots, \alpha_n$ *innerhalb eines Typschemas* $\forall \alpha_1, \ldots, \alpha_n.\tau | C$ *heißen generisch, alle anderen in* $\tau | C$ *vorkommenden Typvariablen spezifisch.*

Def. 5.5.2 (Typumgebung) *Ein Typumgebung $\Gamma \in TE$ ist eine Abbildung von Typvariablen auf eingeschränkte Typschemata. TE bezeichnet die Menge aller Typumgebungen.*

Die Vereinigung zweier Typumgebungen Γ_1 und Γ_2 mit disjunkten Definitionsbereichen ist wie folgt definiert:

$$(\Gamma_1 \cup \Gamma_2)(\alpha) = \begin{cases} \Gamma_2(\alpha), & \text{falls } \Gamma_1(\alpha) \text{ undefiniert} \\ \Gamma_1(\alpha), & \text{falls } \Gamma_2(\alpha) \text{ undefiniert} \end{cases}$$

Zusätzlich definieren wir eine weitere Verknüpfung zweier Typumgebungen Γ_1 und Γ_2, die Γ_2 vorzieht, falls ein Bezeichner zum Definitionsbereich beider Umgebungen gehört. Wir benötigen diese, wenn wir die globale Typumgebung eines Programms um die Belegung der in einem Template auftretenden Attribute erweitern.

$$(\Gamma_1 \cup \Gamma_2)(\alpha) = \begin{cases} \Gamma_2(\alpha), & \text{falls } \Gamma_2(\alpha) \text{ definiert} \\ \Gamma_1(\alpha), & \text{sonst} \end{cases}$$

Der Begriff der generischen Instanz wird auf eingeschränkte Typen erweitert:

Def. 5.5.3 (Generische Instanz)
*Ein eingeschränkter Typ $\tau'|C'$ heißt **generische Instanz** eines Typschemas*

$$\forall \alpha_1, \ldots, \alpha_n . \tau | C,$$

falls eine Substitution σ existiert, so dass

$$\sigma\, \tau \mid \sigma\, C \;=\; \tau' \; |C',$$

mit $\sigma(\alpha_i) = \tau_i$ für alle $i = 1, \ldots, n$, $\sigma(\beta) = \beta$ für alle $\beta \notin \{\alpha_1, \ldots, \alpha_n\}$.

Ein Attribut, das in einem Template T eingeführt wird, ist durch die Vererbung auch in allen Subtemplates von T sichtbar, und besitzt dort denselben Typ wie in T. Um die Typen der Attribute auch innerhalb der Methoden eines Subtemplates sichtbar zu machen, verwenden wir eine Attributtypumgebung:

Def. 5.5.4 (Attributtypumgebung)
*Eine **Attributtypumgebung***

$$\Theta \;:\; \mathit{Templates} \;\rightarrow\; \mathit{TE}$$

ist eine Funktion, die den Namen eins Templates auf eine Typumgebung abbildet.

Nun können wir die Wohldefiniertheit von Funktions- und Templatedefinitionen sowie eines Core ObjectCurry-Programms bestimmen:

Def. 5.5.5 (Wohlgetyptheit von Funktions- und Templatedefinitionen)
Eine Funktionsdefinition

$$f\; t_1 \;\ldots\; t_n \;\; = \;\; e$$

*heißt **wohlgetypt** bzgl. einer Typumgebung Γ, einer Attributtypumgebung Θ und einer Vererbungshierachie $\mathcal{H}^*$, falls die folgenden Bedingungen erfüllt sind:*

- $\Gamma(f) = \forall\alpha_1, \ldots \alpha_m.\tau | C$

- $\Gamma, \Theta, \mathcal{H}^* \vdash \lambda t_1. \cdots .\lambda t_n.e : \tau \,|\, C$
  ist aus den in Tabelle 5.4 und Tabelle 5.5 aufgeführten Typregeln herleitbar

- $\models_{\mathcal{H}^*} C$

Eine Templatedefinition

$$(c, m_1, \ldots, m_n) \texttt{ = } e$$

heißt ***wohlgetypt*** *bzgl. einer Typumgebung* Γ*, einer Attributtypumgebung* Θ *und einer (reflexiven und transitiven) Vererbungshierachie* $\mathcal{H}^*$*, falls gilt:*

- $\Gamma(c) = \tau_0 | C_0,\ \Gamma(m_i) = \forall\alpha_i.\tau_i | C_i$ *für alle* $i = 1, \ldots, n$,

- $\Gamma, \Theta, \mathcal{H}^* \vdash e : (\tau_0 | C_0, \tau_1 | C_1, \ldots, \tau_n | C_n)$
  ist aus den unten aufgeführten Typregeln herleitbar und

- $\models_{\mathcal{H}^*} C$.

Def. 5.5.6 (Wohlgetyptheit eines Core ObjectCurry-Programms)
Ein Core ObjectCurry-Programm heißt ***wohlgetypt****, wenn eine Typumgebung* Γ*, eine Attributtypumgebung* Θ *und eine Vererbungshierachie* $\mathcal{H}^*$ *existieren, bezüglich derer alle Funktions- und Templatedefinitionen des Programms wohlgetypt sind und ferner gilt:*

$$\Gamma(send) = \forall\tau_1, \tau_2 \,.\, \texttt{Message } \tau_1 \rightarrow \texttt{Object } \tau_2 \rightarrow \texttt{Success | } \{\tau_2 \leq \tau_1\}$$
$$\Gamma(new) = \forall\tau_1, \tau_2 \,.\, \texttt{Constructor } \tau_1 \rightarrow \texttt{Object } \tau_2 \rightarrow \texttt{Success | } \{\tau_1 \leq \tau_2\}$$

Innerhalb der Inferenzregeln verwenden wir die Hilfsfunktionen *parents* und *templates*, die die Vorfahren eines Templates bzw. alle Templates einer Vererbungshierachie bestimmen

$$parents(\mathcal{H}^*, name) = \{p \mid (name, p) \in \mathcal{H}^*\}$$

$$templates(\mathcal{H}^*) = \{s \mid (s, p) \in \mathcal{H}^*\} \cup \{p \mid (s, p) \in \mathcal{H}^*\}$$

Mit den in Tabelle 5.4 und Tabelle 5.5 angegebenen Inferenzregeln lässt sich prüfen, ob ein Programm wohlgetypt ist. Dazu muss die Typumgebung Γ bereits die Typen jedes über Funktions- bzw. Templatedefinitionen eingeführten Bezeichners enthalten. Die Attributtypumgebung Θ bildet den Namen jedes Templates auf eine eigene Typumgebung ab. Eine solche Typumgebung enthält die Typen der in dem jeweiligen Template neu definierten Attribute. Die Vererbungshierachie $\mathcal{H}^*$ enthält die Beziehungen zwischen den Templates: $(t_2, t_1) \in \mathcal{H}^*$, falls t_1 (nicht unbedingt direkter) Vorfahre von t_2.

Durch Anwendung der Typregeln lässt sich nun feststellen, ob die gewählten Typumgebungen korrekt sind, und damit das Programm wohlgetypt ist. Die Inferenzregeln leisten keine Typinferenz; einen solchen Algorithmus stellt Kapitel 5.6 vor.

Die Inferenzregeln [Axiom], [Abstraktion], [FreeVars] und [Applikation] unterscheiden sich von den gleichnamigen Regeln des Typsystems für Curry nur durch die Angabe von Typconstraints. Alle innerhalb der einzelnen Teilausdrücke auftretenden Constraints werden

[Axiom] $$\frac{}{\Gamma,\Theta,\mathcal{H}^* \vdash x : \tau|C} \text{ falls } \tau|C \text{ generische Instanz von } \Gamma(x)$$

[Abstraktion] $$\frac{\Gamma[x/\tau|C],\Theta,\mathcal{H}^* \vdash E : \tau'|C'}{\Gamma,\Theta,\mathcal{H}^* \vdash \lambda x.E : \tau \rightarrow \tau'|C'}$$

[FreeVars] $$\frac{\Gamma[x/\tau|C],\Theta,\mathcal{H}^* \vdash E : \tau'|C'}{\Gamma,\Theta,\mathcal{H}^* \vdash \texttt{let}\, x\, \texttt{free in}\, E : \tau'|C \cup C'}$$

[Applikation] $$\frac{\Gamma,\Theta,\mathcal{H}^* \vdash E_1 : \tau_1 \rightarrow \tau_2|C_1 \quad \Gamma,\Theta,\mathcal{H}^* \vdash E_2 : \tau_1|C_2}{\Gamma,\Theta,\mathcal{H}^* \vdash E_1\, E_2 : \tau_2|C_1 \cup C_2}$$

[Template] $$\frac{\begin{array}{lll} name & \in & templates(\mathcal{H}^*) \\ (name, x) & \notin & \mathcal{H}^* \text{ (für alle } x \in templates(\mathcal{H}^*), x \neq name) \\ \Gamma' & = & \Gamma \oplus \Theta(name) \\ \Gamma',\Theta,\mathcal{H}^* & \vdash_I^{name} & I : \tau_0|C_0 \\ \Gamma',\Theta,\mathcal{H}^* & \vdash_M^{name} & M_i : \tau_i|C_i \;\; (i = 1,\ldots,m) \end{array}}{\Gamma,\Theta,\mathcal{H}^* \;\vdash\; \texttt{Template}\, name\; I\; M_1\; \ldots\; M_m \;: (\tau_0|C_0, \tau_1|C_1, \ldots, \tau_m|C_m)}$$

[Subtemplate] $$\frac{\begin{array}{lll} (name_1, name_2) & \in & \mathcal{H}^* \\ (name_2, name_1) & \notin & \mathcal{H}^* \\ \Gamma' & = & \Gamma \oplus \bigcup_{p \in parents(\mathcal{H}^*, name_1)} \Theta(p) \\ \Gamma',\Theta,\mathcal{H}^* & \vdash_{I'}^{\substack{name_1 \\ name_2}} & I' : \tau_0|C_0 \\ p_i & \in & parents(\mathcal{H}^*, name_1) \quad (i = 1,\ldots,m) \\ \Gamma',\Theta,\mathcal{H}^* & \vdash_M^{p_i} & M_i : \tau_i|C_i \quad (i = 1,\ldots,m) \end{array}}{\begin{array}{l}\Gamma,\Theta,\mathcal{H}^* \;\vdash\; \texttt{Template}\, name_1\; \texttt{extends}\; name_2\; I'\; M_1\; \ldots\; M_m \\ \qquad : (\tau_0|C_0, \tau_1|C_1, \ldots, \tau_m|C_m)\end{array}}$$

Tabelle 5.4.: Typregeln für ObjectCurry-Programme (1)

aufgesammelt. Die Überprüfung, ob diese Constraints erfüllbar sind, geschieht außerhalb der Inferenzregeln über die Definition der Wohlgetyptheit eines Programms.

In der Regel [Abstraktion] brauchen die Constraints C der Variablen x nicht zu den Constraints C' des Ausdrucks E hinzugefügt werden: Entweder enthält E eine Applikation, die x enthält. Dann werden die Constraints von x bereits über die Regel [Applikation] aufgesammelt und sind in C' bereits enthalten, d. h. $C \subseteq C'$. Enthält E hingegen keine Applikation mit x, so spielen der Typ und die Constraints von x gar keine Rolle für E.

In den Voraussetzungen der Regel [Template] wird zunächst die Typumgebung Γ um die Typannahmen der Attribute des Templates zu Γ' erweitert. Diese erhalten wir durch Anwendung der Attributtypumgebung auf den Namen des Templates. Mit Hilfe dieser Typumgebung Γ' berechnen wir unter Zuhilfenahme zweier weiterer Inferenzsysteme die eingeschränkten Typen der durch I definierten Konstruktorfunktion und der durch M_i gegebenen Methoden. Des Weiteren wird überprüft, ob in der Vererbungshierachie $\mathcal{H}^*$ kei-

$$[\text{Zuweisung}_1]\quad \frac{\begin{array}{lll}\Gamma,\Theta,\mathcal{H}^* & \vdash & x:\tau|C_1\\ \Gamma,\Theta,\mathcal{H}^* & \vdash & E:\tau|C_2\\ \Gamma,\Theta,\mathcal{H}^* & \vdash_A & A:C_A\end{array}}{\Gamma,\Theta,\mathcal{H}^* \vdash_A x:=E,A : C_1\cup C_2\cup C_A}$$

$$[\text{Zuweisung}_2]\quad \frac{}{\Gamma,\Theta,\mathcal{H}^* \vdash_A \epsilon : \emptyset}$$

$$[\text{Init}]\quad \frac{\Gamma,\Theta,\mathcal{H}^* \ \vdash_A \ A:C}{\Gamma,\Theta,\mathcal{H}^* \vdash_I^{name} A : \texttt{Constructor}\ name|C}$$

$$[\text{Init'}]\quad \frac{\begin{array}{lll}\Gamma,\Theta,\mathcal{H}^* & \vdash & E:\texttt{Constructor}\ name_2|\epsilon\\ \Gamma,\Theta,\mathcal{H}^* & \vdash_A & A:C\end{array}}{\Gamma,\Theta,\mathcal{H}^* \vdash_{I'}^{\overset{name_1}{name_2}} E,A : \texttt{Constructor}\ name_1|C}$$

$$[\text{Methode}]\quad \frac{\begin{array}{lll}v & & \text{neue Typvariable}\\ \Gamma,\Theta,\mathcal{H}^* & \vdash & E:\texttt{Success}|C\\ \Gamma,\Theta,\mathcal{H}^* & \vdash_A & A:C'\end{array}}{\Gamma,\Theta,\mathcal{H}^* \vdash_M^{name} E\Rightarrow A : \texttt{Message}\ v|\{v\leq name\}\cup C\cup C'}$$

$$[\text{Abstraktion}_{\mathcal{X}}]\quad \frac{\Gamma,\Theta,\mathcal{H}^*[x/\tau|C]\vdash_{\mathcal{X}}^n X:\tau'|C'}{\Gamma,\Theta,\mathcal{H}^*\vdash_{\mathcal{X}}^n \lambda x.X:\tau\to\tau'|C'}\qquad \mathcal{X}\in\{I,I',M\}$$

Tabelle 5.5.: Typregeln für ObjectCurry-Programme (2)

ne echten Vorfahren zu dem definierten Template enthalten sind, sondern das Template selbst sein einziger Vorfahre ist.

Da Γ und damit auch Γ' alle Typen der im Programm definierten Bezeichner und damit auch die Typen der Nachrichtenbezeichner enthält, können innerhalb von Methoden eines Templates auch die Nachrichten dieses Templates verwendet werden. Es wird keine zusätzliche Regel zur Rekursion benötigt.

Die Regel [Subtemplate] entspricht der Regel [Template], enthält aber folgende Unterschiede:

- Die Typumgebung Γ' muss neben den Typannahmen der neu definierten Attribute auch die aller geerbten Attribute enthalten. Diese stammen nicht nur vom direkten Vorfahren, sondern auch von dessen Vorfahren ab. Wir verwenden die oben eingeführte Hilfsfunktion *parents*, um die Namen der Vorfahren zu ermitteln.

- I' enthält den Aufruf der Konstruktorfunktion des Vorfahren. Es muss sichergestellt werden, dass dieser den Typ `Constructor` $name_2$ besitzt, wobei $name_2$ der Name des Vorfahren ist. Diese Überprüfung erfolgt durch Verwendung von $\vdash_{I'}$ anstelle von $\vdash_I$.

- Des Weiteren muss sichergestellt werden, dass $(name_1, name_2)$ ein Element der Vererbungshierachie $\mathcal{H}^*$ ist, also $name_1$ ein Untertyp von $name_2$ ist, und dass der umgekehrte Fall nicht gilt. Da $\mathcal{H}^*$ eine transitive und reflexive Funktion ist, enthält $\mathcal{H}^*$ auch die Elemente $(name_2, name_2)$ und $(name_1, name_1)$, sowie $(name_1, p)$ für alle Vorfahren p von $name_1$.
- Bei der Überprüfung der Typen der Methoden lassen wir auch zu, dass es sich um eine Methode eines Vorfahren handelt. Dies ist nötig, wenn die jeweilige Methode in dem Subtemplate redefiniert wurde.

Die Inferenzregel [Zuweisung$_1$] aus Tabelle 5.5 bewirkt, dass in einer Zuweisung der Form x `:=` E der Typ des Attributs x gleich dem Typ des Ausdrucks E ist. Die Regel [Zuweisung$_2$] ist für die leere Folge von Zuweisungen ϵ zuständig.

Die Regel [Init] überprüft den Typ der Konstruktorfunktion. Dabei muss der Name des Templates als zusätzlicher Parameter `name` mitgeführt werden. [Init'] verhält sich wie [Init], prüft aber zusätzlich, ob E ein gültiger Aufruf der Konstruktorfunktion des Vorfahren ist. Dazu muss auch der Name des Vorfahren $(name_2)$ mitgeführt werden.

Zum Typen von Methoden dient die Regel [Methode]. Sie prüft, ob der Ausdruck E einer Methode $E \Rightarrow A$ ein Constraint ist, also den Typ `Success` besitzt, und sammelt die durch die Zuweisungen A entstehenden Constraints auf.

Um die Lambda-Abstraktionen, die sowohl über initialen Zuweisungen I bzw. I', als auch über Methoden M erlaubt sind, zu behandeln, existiert die generische Regel [Abstraktion$_\mathcal{X}$]. $\mathcal{X}$ steht hierbei für I, I' oder M. Bis auf das Mitführen des zusätzlichen Parameters n entspricht diese Regel der gewöhnlichen Abstraktionsregel über Ausdrücken.

Beispiel

Als Beispiel betrachten wir das Template `MaxCounter`. Wir wollen zeigen, dass die Templatedefinition von `MaxCounter` aus Beispiel 5.4.2 wohlgetypt ist bezüglich der folgenden intuitiv gewählten Typumgebung Γ, Attributtypumgebung Θ und Vererbungshierachie $\mathcal{H}^*$:

$$
\begin{array}{lcl}
\Gamma(\texttt{counter}) & = & \forall.\texttt{Int} \rightarrow \texttt{Constructor Counter} \mid \emptyset \\
\Gamma(\texttt{maxCounter}) & = & \forall.\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Constructor MaxCounter} \mid \emptyset \\
\Gamma(\texttt{Inc}) & = & \forall\alpha.\texttt{Message}\ \alpha \mid \alpha \leq \texttt{Counter} \\
\Gamma(\texttt{Set}) & = & \forall\alpha.\texttt{Int} \rightarrow \texttt{Message}\ \alpha \mid \alpha \leq \texttt{Counter} \\
\Gamma(\texttt{SetMax}) & = & \forall\alpha.\texttt{Int} \rightarrow \texttt{Message}\ \alpha \mid \alpha \leq \texttt{MaxCounter} \\
\Gamma(\texttt{send}) & = & \forall\alpha,\beta.\texttt{Message}\ \alpha \rightarrow \texttt{Object}\ \beta \rightarrow \texttt{Sucess} \mid \beta \leq \alpha \\
\Gamma(\texttt{new}) & = & \forall\alpha,\beta.\texttt{Constructor}\ \alpha \rightarrow \texttt{Object}\ \beta \rightarrow \texttt{Sucess} \mid \alpha \leq \beta \\
\Gamma(\texttt{success}) & = & \forall.\texttt{Success} \mid \emptyset \\
 & & \\
\Theta(\texttt{Counter}) & = & \texttt{x} \mapsto \texttt{Int} \\
\Theta(\texttt{MaxCounter}) & = & \texttt{max} \mapsto \texttt{Int} \\
 & & \\
\mathcal{H}^* & = & \{(\texttt{Counter}, \texttt{Counter}), (\texttt{MaxCounter}, \texttt{Counter}), \\
 & & (\texttt{MaxCounter}, \texttt{MaxCounter})\}
\end{array}
$$

Das Template `MaxCounter` ist typkorrekt, wenn unter diesen Annahmen mit Hilfe der Regeln der Tabellen 5.4 und 5.5 die Typen der Konstruktorfunktion und der Methoden herleitbar sind:

$$
\begin{array}{lll}
\Gamma, \Theta, \mathcal{H}^* & \vdash & \texttt{Template MaxCounter extends Counter} \\
& & \lambda\texttt{i} \; . \; \lambda\texttt{m} \; . \; \texttt{counter i}, \texttt{max} := \texttt{m} \\
& & \texttt{success} \Rightarrow \texttt{x} := \texttt{if x<max then x+1 else x} \\
& & \lambda\texttt{v} \; . \; \texttt{success} \Rightarrow \texttt{max} := \texttt{v} \\
& : & (\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Constructor MaxCounter} \mid \emptyset, \\
& & \texttt{Message } \alpha \mid \alpha \leq \texttt{Counter}, \\
& & \texttt{Int} \rightarrow \texttt{Message } \beta \mid \beta \leq \texttt{MaxCounter})
\end{array}
$$

Die Anwendung der Regel [Subtemplate] von unten nach oben führt zu den folgenden Bedingungen:

$$
\begin{array}{lll}
(\texttt{MaxCounter}, \texttt{Counter}) & \in & \mathcal{H}^* \\
(\texttt{Counter}, \texttt{MaxCounter}) & \notin & \mathcal{H}^*
\end{array}
$$

Diese sind erfüllt, wie sich leicht überprüfen lässt.

Im weiteren Verlauf benötigen wir eine neue Typumgebung Γ'. Dazu wird die Attributtypumgebung auf alle Vorfahren von `MaxCounter` angewendet, und die sich ergebenden Typumgebungen zu Γ hinzugefügt:

$$
\begin{array}{lll}
\Gamma' & = & \Gamma \oplus (\Theta(\texttt{MaxCounter}) \cup \Theta(\texttt{Counter})) \\
& = & \Gamma \oplus \{\texttt{x} \mapsto \texttt{Int}, \texttt{max} \mapsto \texttt{Int}\}
\end{array}
$$

$\oplus$ hat in diesem konkreten Fall dieselbe Wirkung wie $\cup$, da kein Bezeichner im Definitionsbereich beider Typumgebungen vorkommt.

Zur Erfüllung der Regel [Subtemplate] bleibt zu zeigen, dass sich die Typen der Konstruktorfunktion und der beiden Methoden von `MaxCounter` wie folgt herleiten lassen:

$$
\begin{array}{llll}
(a) & \Gamma', \Theta, \mathcal{H}^* & \vdash_{\Gamma'}^{\texttt{Counter}}{}^{\texttt{MaxCounter}} & \lambda\texttt{i} \; . \; \lambda\texttt{m} \; . \; \texttt{counter i}, \texttt{max} := \texttt{m} \\
& & & : \texttt{Int -> Int -> Constructor MaxCounter} \\
(b) & \Gamma', \Theta, \mathcal{H}^* & \vdash_{M}^{\texttt{Counter}} & \texttt{success} \Rightarrow x := \texttt{if x<max then x+1 else x} \\
& & & : \texttt{Message } \alpha \mid \alpha \leq \texttt{Counter} \\
(c) & \Gamma', \Theta, \mathcal{H}^* & \vdash_{M}^{\texttt{MaxCounter}} & \lambda v \; . \; \texttt{success} \Rightarrow \texttt{max} := \texttt{v} \\
& & & : \texttt{Int} \rightarrow \texttt{Message } \beta \mid \beta \leq \texttt{MaxCounter}
\end{array}
$$

Wir beginnen mit der Herleitung von (a). Dazu verwenden wir zweimal die Abstraktionsregel und anschließend die Regel [Init'] (von unten nach oben):

$$\dfrac{\dfrac{\dfrac{\dfrac{(1)}{\Gamma'',\Theta,\mathcal{H}^* \vdash \texttt{counter}\ \texttt{i} : \texttt{Constructor Counter}} \qquad \dfrac{(2)}{\Gamma'',\Theta,\mathcal{H}^* \vdash_A \texttt{max} := \texttt{m} : \emptyset}}{\Gamma'[\texttt{i/Int},\texttt{m/Int}],\Theta,\mathcal{H}^* \vdash_{I'}^{\substack{\texttt{Counter}\\ \texttt{MaxCounter}}} \begin{array}{l} \texttt{counter}\ \texttt{i},\texttt{max} := \texttt{m} \\ : \texttt{Constructor MaxCounter} \end{array}}}{\Gamma'[\texttt{i/Int}],\Theta,\mathcal{H}^* \vdash_{I'}^{\substack{\texttt{Counter}\\ \texttt{MaxCounter}}} \begin{array}{l} \lambda\texttt{m}\ .\ \texttt{counter}\ \texttt{i},\texttt{max} := \texttt{m} \\ : \texttt{Int -> Constructor MaxCounter} \end{array}}}{\Gamma',\Theta,\mathcal{H}^* \vdash_{I'}^{\substack{\texttt{Counter}\\ \texttt{MaxCounter}}} \begin{array}{l} \lambda\texttt{i}\ .\ \lambda\texttt{m}\ .\ \texttt{counter}\ \texttt{i},\texttt{max} := \texttt{m} \\ : \texttt{Int -> Int -> Constructor MaxCounter} \end{array}}$$

Um Platz zu sparen, definieren wir $\Gamma'' \equiv \Gamma''[\texttt{i/Int},\texttt{m/Int}]$.

(1) lässt sich mit Hilfe der Regeln [Applikation] und [Axiom] herleiten:

$$\dfrac{\dfrac{\overline{\Gamma''(\texttt{counter}) = \forall.\texttt{Int -> Constructor Counter}}}{\Gamma'',\Theta,\mathcal{H}^* \vdash \texttt{counter} : \texttt{Int} \to \texttt{Constructor Counter}} \qquad \dfrac{\overline{\Gamma''(\texttt{i}) = \forall.\texttt{Int}}}{\Gamma'',\Theta,\mathcal{H}^* \vdash \texttt{i} : \texttt{Int}}}{\Gamma'',\Theta,\mathcal{H}^* \vdash \texttt{counter}\ \texttt{i} : \texttt{Constructor Counter}}$$

Zur Herleitung von (2) wird die Regel [Zuweisung$_1$] verwendet. Die sich daraus ergebenden Bedingungen lassen sich mit der Regel [Axiom] bzw. [Zuweisung$_2$] zeigen.

$$\begin{array}{lll} \Gamma'',\Theta,\mathcal{H}^* & \vdash & \texttt{max} : \texttt{Int} \\ \Gamma'',\Theta,\mathcal{H}^* & \vdash & \texttt{m} : \texttt{Int} \\ \Gamma'',\Theta,\mathcal{H}^* & \vdash_A & \epsilon : \emptyset \end{array}$$

Als nächstes leiten wir (c) her.

$$\dfrac{\dfrac{\Gamma'[\texttt{v/Int}],\Theta,\mathcal{H}^* \vdash \texttt{success} : \texttt{Success} \qquad \dfrac{\begin{array}{lll} \Gamma'',\Theta,\mathcal{H}^* & \vdash & \texttt{max} : \texttt{Int} \\ \Gamma'',\Theta,\mathcal{H}^* & \vdash & \texttt{v} : \texttt{Int} \\ \Gamma'',\Theta,\mathcal{H}^* & \vdash_A & \epsilon : \emptyset \end{array}}{\Gamma'[\texttt{v/Int}],\Theta,\mathcal{H}^* \vdash_A \texttt{max} := \texttt{v},\epsilon : \emptyset}}{\Gamma'[\texttt{v/Int}],\Theta,\mathcal{H}^* \vdash_M^{\texttt{MaxCounter}} \begin{array}{l} \texttt{success} \Rightarrow \texttt{max} := \texttt{v} \\ : \texttt{Message}\ \beta \mid \{\beta \leq \texttt{MaxCounter}\} \end{array}}}{\Gamma',\Theta,\mathcal{H}^* \vdash_M^{\texttt{MaxCounter}} \begin{array}{l} \lambda v\ .\ \texttt{success} \Rightarrow \texttt{max} := \texttt{v} \\ : \texttt{Int} \to \texttt{Message}\ \beta \mid \{\beta \leq \texttt{MaxCounter}\} \end{array}}$$

Wir verzichten an dieser Stelle auf die Herleitung von (b), da diese ähnlich wie (c) verläuft.

5.6. Typinferenz

In diesem Kapitel präsentieren wir einen **Typinferenzalgorithmus** für Curry. Er basiert auf dem von Kaes [Kae92] vorgestellten Algorithmus $\mathcal{D}$, konnte aber, da wir Untertypenbildung nur auf Objekten und Nachrichten zulassen, erheblich vereinfacht werden.

Der Algorithmus besteht aus zwei Schritten: Zunächst werden mit Hilfe der unten angegebenen Funktion $\mathcal{E}$ die eingschränkten Typausdrücke jeder im Programm auftretenden Funktions- und Templatedefinition bestimmt. Bei rekursiven und wechselseitig rekursiven Definitionen ist es zunächst erforderlich, die Typumgebung Γ vorher in der Weise zu erweitern, dass jedem Bezeichner eine neue Typvariable zugeordnet wird. Nach Anwendung von $mathcalE$ müssen diese Variablen mit den berechneten eingeschränkten Typen unifiziert werden. Für die genaue Vorgehensweise sei auf [Jon99] verwiesen. Die so berechneten eingeschränkten Typen enthalten Constraints. Diese werden in einem zweiten Schritt auf Erfüllbarkeit überprüft.

5.6.1. Typinferenz

Die Funktion $\mathcal{E}$ berechnet den eingeschränkten Typausdruck eines beliebigen Ausdrucks, bei dem es sich auch um ein Template handeln kann. Vor Anwendung von $\mathcal{E}$ muss jedoch

$\mathcal{E}[\![x]\!](\Gamma, \Theta, \mathcal{H}^*) = (\emptyset, \tau|C)$
  falls $\tau|C$ Instanz von $\Gamma(x)$

$\mathcal{E}[\![\lambda x.E]\!](\Gamma, \Theta, \mathcal{H}^*) = (\sigma, (\sigma\alpha) \to \tau|\sigma\ C)$
  falls Substitution σ existiert und α frische Typvariable, so dass
  $(\sigma, \tau|C) = \mathcal{E}[\![E]\!](\Gamma[x/\alpha])$

$\mathcal{E}[\![\texttt{let}\ x\ \texttt{free in}\ E]\!](\Gamma, \Theta, \mathcal{H}^*) = (\sigma, \tau|\sigma\ C)$
  falls Substitution σ existiert und α frische Typvariable, so dass
  $(\sigma, \tau|C) = \mathcal{E}[\![E]\!](\Gamma[x/\alpha])$

$\mathcal{E}[\![E_1\ E_2]\!](\Gamma, \Theta, \mathcal{H}^*) = (\sigma \circ \sigma_2 \circ \sigma_1, (\sigma\alpha)|\sigma\ (\sigma_2\ C_1\ \cup C_2))$
  falls Substitutionen $\sigma_1, \sigma_2, \sigma_3$ existieren und α frische Typvariable, so dass
  $(\sigma_1, \tau_1|C_1) = \mathcal{E}[\![E_1]\!](\Gamma, \Theta, \mathcal{H}^*)$
  $(\sigma_2, \tau_2|C_2) = \mathcal{E}[\![E_2]\!](\sigma_1\ \Gamma, \Theta, \mathcal{H}^*)$
  $\sigma = mgu(\tau_1, \tau_2 \to \alpha)$

$\mathcal{E}[\![\texttt{Template}\ t\ I\ M_1\ \dots\ M_m]\!](\Gamma, \Theta, \mathcal{H}^*) = (\sigma, (\tau_0|C_0, \tau_1|C_1, \dots, \tau_n|C_n))$
  falls Substitutionen $\sigma_0, \sigma_1, \dots \sigma_m$ existieren
  $\Gamma' = \Gamma \oplus \Theta(t)$
  $(\sigma_0, \tau_0|C_0) = \mathcal{I}[\![I]\!](t, \Gamma', \Theta, \mathcal{H}^*)$
  $(\sigma_i, \tau_i|C_i) = \mathcal{M}[\![M_i]\!](t, \Gamma', \Theta, \mathcal{H}^*)$ für alle $i = 1, \dots, n$

$\mathcal{E}[\![\texttt{Template}\ t_1\ \texttt{extends}\ t_2\ I'\ M_1\ \dots\ M_m]\!](\Gamma, \Theta, \mathcal{H}^*) = (\sigma, (\tau_0|C_0, \tau_1|C_1, \dots, \tau_n|C_n))$
  falls Substitutionen $\sigma_0, \sigma_1, \dots \sigma_m$ existieren
  $\Gamma' = \Gamma \oplus \bigcup_{p \in parents(\mathcal{H}^*, t_2)} \Theta(p)$
  $(\sigma_0, \tau_0|C_0) = \mathcal{I}'[\![I']\!](t_1, t_2, \Gamma', \Theta, \mathcal{H}^*)$
  $(\sigma_i, \tau_i|C_i) = \begin{cases} \mathcal{M}[\![M_i]\!](t, \Gamma', \Theta, \mathcal{H}^*) & \text{falls } \Gamma(t) = \cdots \to \texttt{Message}\ t \\ \mathcal{M}[\![M_i]\!](t_1, \Gamma', \Theta, \mathcal{H}^*) & \text{sonst} \end{cases}$
  für alle $i = 1, \dots, n$

Tabelle 5.6.: Typinferenzalgorithmus für ObjectCurry (1) zunächst eine Typumgebung Γ, eine Vererbungshierachie $\mathcal{H}^*$ und eine Attributtypumgebung Θ aufgebaut werden. Dazu durchlaufen wir in einem seperaten Durchgang alle Templatedefinitionen. Wir starten mit einer leeren Attributtypumgebung Θ und erweitern Θ bei jedem Template, so dass Θ den Namen des Templates auf eine Typumgebung abbildet. Diese Typumgebung bildet jedes Attribut des Templates auf eine neue Typvariable ab. Gleichzeitig bauen wir die Vererbungshierachie $\mathcal{H}^*$ auf. In der Typumgebung Γ müssen allen Bezeichnern rekursiver bzw. wechselseitig rekursiver Funktionen eine neue Typvariable zugeordnet werden. Das Gleiche gilt für die Namen von Konstruktoren und Methoden, wenn diese rekursiv in dem Template verwendet werden, in dem sie definiert werden. Für eine detalliertere Beschreibung zur Behandlung wechselseitig rekursiver Funktionen verweise ich auf [Jon99] sowie [Thi94].

$$
\begin{array}{l}
\mathcal{I}[\![A]\!](t,\Gamma,\Theta,\mathcal{H}^*) = (\sigma, \texttt{Constructor}\; t|C) \\
\quad \text{falls Substitution } \sigma \text{ existiert, so dass} \\
\quad (\sigma, C) = \mathcal{A}[\![A]\!](\Gamma,\Theta,\mathcal{H}^*) \\[4pt]
\mathcal{I}[\![\lambda x.I]\!](t,\Gamma,\Theta,\mathcal{H}^*) = (\sigma, (\sigma\alpha) \to \tau|C) \\
\quad \text{falls Substitution } \sigma \text{ existiert, und } \alpha \text{ frische Typvariable, so dass} \\
\quad (\sigma, \tau|C) = \mathcal{I}[\![I]\!](t,\Gamma[x/\alpha]) \\[4pt]
\mathcal{I}'[\![E,A]\!](t_1,t_2,\Gamma,\Theta,\mathcal{H}^*) = (\sigma \circ \sigma_E, \texttt{Constructor}\; t_1|C \cup C_E) \\
\quad \text{falls Substitution } \sigma \text{ existiert, so dass} \\
\quad (\sigma, C) = \mathcal{A}[\![A]\!](\Gamma,\Theta,\mathcal{H}^*) \\
\quad (\sigma_E, \texttt{Constructor}\; t_2|C_E) = \mathcal{E}[\![E]\!](\Gamma,\Theta,\mathcal{H}^*) \\[4pt]
\mathcal{I}'[\![\lambda x.I']\!](t_1,t_2,\Gamma,\Theta,\mathcal{H}^*) = (\sigma, (\sigma\alpha) \to \tau|C) \\
\quad \text{falls Substitution } \sigma \text{ existiert, und } \alpha \text{ frische Typvariable, so dass} \\
\quad (\sigma, \tau|C) = \mathcal{I}'[\![I']\!](t_1,t_2,\Gamma[x/\alpha]) \\[4pt]
\mathcal{M}[\![M \Rightarrow A]\!](\tau,\Gamma,\Theta,\mathcal{H}^*) = (\sigma_2 \circ \sigma_1, \sigma_2(\texttt{Message}\; \alpha)|(\sigma_2 C_1 \cup C_2 \cup \{\alpha \le \tau\}) \\
\quad \text{falls Substitutionen } \sigma_1, \sigma_2 \text{ existieren und } \alpha\text{, frische Typvariableso dass} \\
\quad (\sigma_1, \texttt{Success}|C_1) = \mathcal{E}[\![E]\!](\Gamma,\Theta,\mathcal{H}^*) \\
\quad (\sigma_2, C_2) = \mathcal{A}[\![A]\!](\sigma_1\Gamma,\Theta,\mathcal{H}^*) \\[4pt]
\mathcal{M}[\![\lambda x.M]\!](\tau,\Gamma,\Theta,\mathcal{H}^*) = (\sigma, (\sigma\alpha) \to \tau|C) \\
\quad \text{falls Substitution } \sigma \text{ existiert und } \alpha \text{ frische Typvariable, so dass} \\
\quad (\sigma, \tau|C) = \mathcal{M}[\![M]\!](\tau,\Gamma[x/\alpha]) \\[4pt]
\mathcal{A}[\![x := E, A]\!](\Gamma,\Theta,\mathcal{H}^*) = (\sigma_3 \circ \sigma \circ \sigma_2 \circ \sigma_1, (\sigma_3 \circ \sigma \circ \sigma_2)C_1 \cup (\sigma_3 \circ \sigma)C_2 \cup C_3) \\
\quad \text{falls Substitutionen } \sigma_1, \sigma_2, \sigma_3, \sigma \text{ existieren, so dass} \\
\quad (\sigma_1, \tau_1|C_1) = \mathcal{E}[\![x]\!](\Gamma,\Theta,\mathcal{H}^*) \\
\quad (\sigma_2, \tau_2|C_2) = \mathcal{E}[\![E]\!](\Gamma,\Theta,\mathcal{H}^*) \\
\quad \sigma = mgu(\tau_1, \tau_2) \\
\quad (\sigma_3, C_3) = \mathcal{A}[\![A]\!]((\sigma \circ \sigma_2 \circ \sigma_1)\Gamma,\Theta,\mathcal{H}^*) \\[12pt]
\mathcal{A}[\![\epsilon]\!](\Gamma,\Theta,\mathcal{H}^*) = (\emptyset, \emptyset)
\end{array}
$$

Tabelle 5.7.: Typinferenzalgorithmus für ObjectCurry (2)

$$[\text{Transitiv}] \quad \frac{\{\tau_1 \leq \tau_2\, \tau_2 \leq \tau_3\} \cup C}{\{\tau_1 \leq \tau_2, \tau_2 \leq \tau_3, \tau_1 \leq \tau_3\} \cup C} \text{ falls } (\tau_1 \leq \tau_3) \notin C$$

Tabelle 5.8.: Erzeugung der transitiven Hülle

Anschließend können mit Hilfe der Funktion $\mathcal{E}$ nacheinander die eingeschränkten Typen der im Programm definierten Funktionen und Templates berechnet werden. Handelt es sich um eine rekursive Funktion, so muss der Eintrag in Γ mit dem berechneten Typen unifiziert, und die brechneten Constraints hinzugefügt werden. Ansonsten genügt es, Γ in der Weise zu erweitern, dass es den Namen der Funktion auf den soeben berechneten eingeschränkten Typ abbildet. Das Vorgehen für Templatedefinitionen ist analog.

Nachdem auf diese Weise die eingeschränkten Typen aller definierten Funktionen, Konstruktoren und Methoden inferiert wurden, muss getestet werden, ob die dabei entstandenen Constraints erfüllbar sind.

5.6.2. Erfüllbarkeitstest

Eine Menge von Subtypconstraints C ist erfüllbar bzgl. einer Vererbungshierachie $\mathcal{H}^*$, falls eine Substitution σ existiert, so dass $\sigma \models_{\mathcal{H}^*} c$ für alle $c \in C$. Wir werden in diesem Abschnitt ein Verfahren vorstellen, mit dem eine Menge von Subtypcontraints auf Erfüllbarkeit getest werden kann, ohne dass eine Substitution der auftretenden Variablen angeben werden muss.

Gegeben sei eine Menge von Subtypconstraints C, die es zu untersuchen gilt. Die Constraints aus C enthalten Typvariablen und Templatetypen. Im Folgenden werden wir Typvariablen immer mit den Kleinbuchstaben $a, b, \ldots$ bzw. $t_1, t_2, \ldots$ und Templatetypen mit den Großbuchstaben A, B, D, X kennzeichnen.

Die Menge C ist nicht erfüllbar, wenn sie einen Constraint der Form $A \leq B$ enthält, und im aktuellen Programm B kein Vorfahre von A ist, also $(A, B) \notin \mathcal{H}^*$ gilt, denn der Constraint $A \leq B$ ist bzgl. $\mathcal{H}^*$ nicht erfüllbar, und damit ist auch C nicht erfüllbar. C ist ebenfalls nicht erfüllbar, wenn eine transitive Kette aus Templatevariablen zwischen zwei Templatetypen A und B mit $(A, B) \notin \mathcal{H}^*$ existiert:

$$(A \leq t_1) \in C, (t_1 \leq t_2) \in C, \ldots, (t_{n-1}, t_n) \in C, (t_n \leq B) \in C$$

In diesem Fall lässt sich nämlich keine Substitution finden, die alle diese Constraints bezüglich $\mathcal{H}^*$ erfüllt. Das Vorkommen einer solchen Kette in C lässt sich sehr einfach überprüfen, indem man die transitive Hülle zu C bildet: Existiert eine Kette zwischen A und B, dann enthält die transitive Hülle ein Element $A \leq B$. Aus diesem Grund betrachten wir im Folgenden immer die transitive Hülle zu C und bezeichnen sie mit C^+. Sie lässt sich mit Hilfe der Regel aus Tabelle 5.8 berechnen, indem diese sooft wie möglich auf C angewendet wird.

Um eine Menge von Constraints C auf Erfüllbarkeit zu untersuchen, starten wir mit der transitiven Hülle C^+ und entfernen nacheinander nach bestimmten Regeln jeweils ein Ele-

$$[\text{Axiom}] \quad \frac{}{\mathcal{H}^* \vdash_1 \{\}}$$

$$[\text{Übergang}] \quad \frac{\mathcal{H}^* \vdash_i C}{\mathcal{H}^* \vdash_{i+1} C} \quad i \in \{1, 2, 3\}$$

$$[\text{Templates}] \quad \frac{\mathcal{H}^* \vdash_1 C \quad (A, B) \in \mathcal{H}^*,}{\mathcal{H}^* \vdash_1 \{A \leq B\} \cup C}$$

$$[\text{Unterklasse}_1] \quad \frac{\mathcal{H}^* \vdash_2 C \quad \forall X \in \mathit{templates}(\mathcal{H}^*) : \ (a \leq X) \notin C}{\mathcal{H}^* \vdash_2 \{a \leq A\} \cup C}$$

$$[\text{Unterklasse}_2] \quad \frac{\mathcal{H}^* \vdash_3 \{a \leq B\} \cup C \quad (B, A) \in \mathcal{H}^*}{\mathcal{H}^* \vdash_3 \{a \leq A\} \cup \{a \leq B\} \cup C}$$

$$[\text{Oberklasse}] \quad \frac{\mathcal{H}^* \vdash_4 C}{\mathcal{H}^* \vdash_4 \{A \leq a\} \cup C}$$

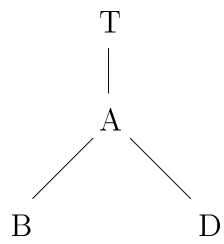
$$[\text{Variablen}] \quad \frac{\mathcal{H}^* \vdash_5 C}{\mathcal{H}^* \vdash_5 \{a \leq b\} \cup C}$$

Die Symbole a, b bezeichnen Typvariaben, A, B, X Templatetypen, C eine Menge von Constraints und $\mathcal{H}^*$ die transitive Hülle der Vererbungshierachie.

Tabelle 5.9.: Erfüllbarkeitstest für Constraints

ment aus dieser Menge. Gelingt es, alle Elemente aus C^+ zu entfernen, ist C erfüllbar. Ist jedoch keine Regel mehr anwendbar, so ist C nicht erfüllbar. Die Regeln, mit denen Elemente aus C^+ entfernt werden dürfen, sind durch die Inferenzregeln aus Tabelle 5.9 bestimmt. Die Regeln sind dabei von unten nach oben anzuwenden. Es ist leicht zu sehen, dass jede der Regeln einen Constraint entfernt. Um den Korrektheitsbeweis dieser Regeln zu vereinfachen, lassen wir nicht die beliebige Anwendung zu, sondern legen über die Indizes an den Symbolen $\vdash$ eine Reihenfolge fest.

Betrachten wir zunächst einige Beispiele für erfüllbare und nicht-erfüllbare Constraintmengen. Für alle Beispiele wählen wir die Vererbungshierachie $\mathcal{H}^*$ wie folgt:



$$\mathcal{H}^* = \{(A, A), (B, B), (D, D), (T, T), (A, T), (B, T), (D, T), (B, A), (D, A)\}$$

Zunächst betrachten wir eine bzgl. $\mathcal{H}^*$ erfüllbare Constraintmenge C_1:

$$C_1 = \{B \leq b, a \leq b, b \leq A\}$$

Die transitive Hülle lautet

$$C_1^+ = \{B \leq b, a \leq b, b \leq A, B \leq A\}$$

$\mathcal{H}^* \vdash_5 C_1^+$ lässt sich mit den Inferenzregeln aus Tabelle 5.9 erzeugen:

$$\begin{array}{c}
\hline
\mathcal{H}^* \vdash_1 \{\} \\
\hline
\mathcal{H}^* \vdash_1 \{B \leq A\} \\
\hline
\mathcal{H}^* \vdash_2 \{B \leq A\} \\
\hline
\mathcal{H}^* \vdash_2 \{a \leq A, B \leq A\} \\
\hline
\mathcal{H}^* \vdash_3 \{a \leq A, B \leq A\} \\
\hline
\mathcal{H}^* \vdash_4 \{a \leq A, B \leq A\} \\
\hline
\mathcal{H}^* \vdash_4 \{B \leq b, a \leq A, B \leq A\} \\
\hline
\mathcal{H}^* \vdash_5 \{B \leq b, a \leq A, B \leq A\} \\
\hline
\mathcal{H}^* \vdash_5 \{b \leq a, B \leq b, a \leq A, B \leq A\} = C_1^+
\end{array}$$

Liest man die Regeln von unten nach oben, so sieht man, dass sich alle Constraints aus C^+ auf diese Weise entfernen lassen.

Als erstes unerfüllbares Beispiel wählen wir eine Menge von Constraints, die eine transitive Kette enthält:

$$C_2 = \{B \leq t_1, t_1 \leq t_2, t_2 \leq D\}$$

Durch Bildung der transitiven Hülle erhalten wir

$$C_2^+ = \{B \leq t_1, t_1 \leq t_2, t_2 \leq D, t_1 \leq D, B \leq D\}$$

Der Constraint $B \leq D$ lässt sich nicht entfernen, da $(B, D) \notin \mathcal{H}^*$.
Als weiteres Beispiel betrachten wir

$$C_3 = \{t \leq B, t \leq D\}$$

C_3 ist gleich seiner transitiven Hülle C_3^+. Keiner der beiden Constraints lässt sich entfernen, da weder die Regel [Unterklasse$_1$] noch die Regel [Unterklasse$_2$] anwendbar ist.

Zum Abschluss möchten wir formal beweisen, dass eine Menge von Constraints C genau dann erfüllbar ist, wenn wir mit Hilfe der Regeln aus Tabelle 5.9 aus der leeren Menge die Menge C^+ konstruieren können. Dazu formulieren wir den folgenden Satz:

Satz 5.6.1
Sei C eine Menge von Constraints und $\mathcal{H}^*$ eine Vererbungshierachie. Dann gilt:

$$\models_{\mathcal{H}^*} C \Longleftrightarrow \mathcal{H}^* \vdash_5 C^+$$

Für den Beweis von Satz 5.6.1 benötigen wir den folgenden Satz:

Satz 5.6.2
Eine Menge C ist genau dann erfüllbar bzgl. einer Vererbungshierachie $\mathcal{H}^*$, wenn ihre reflexive und transitive Hülle C^+ erfüllbar ist bzgl. $\mathcal{H}^*$:

$$\models_{\mathcal{H}^*} C \quad \Longleftrightarrow \quad \models_{\mathcal{H}^*} C^*$$

Beweis:

Die Richtung $\Longrightarrow$ ist trivial, da $C \subseteq C^*$

Für die andere Richtung $\Longleftarrow$ betrachten wir die Definition der transitiven Hülle von C. Dabei sei M die Menge aller Typvariablen und Templatetypen der in C vorkommenden Constraints:

$$\begin{aligned}
&C^* := \bigcup_{n=0}^{\infty} C^n \\
&C^0 := \{a \leq a | a \in M\} \\
&C^{n+1} := C \circ C^n \\
&(a_1, a_2) \in (C_1 \circ C_2) :\Leftrightarrow \exists a \in M : (a_1 \leq a) \in C_1 \text{ und } (a \leq a) \in C_2
\end{aligned}$$

Wir zeigen $\models_{\mathcal{H}^*} C \Longrightarrow \models_{\mathcal{H}^*} \bigcup_{n=0}^{\infty} C^n$ durch vollständige Induktion über n.

Induktionsbeginn $n = 0$:

$$\begin{aligned}
\models_{\mathcal{H}^*} C \quad &\Longrightarrow \quad \exists \sigma : \forall c \in C : \sigma \models_{\mathcal{H}^*} c \\
&\Longrightarrow \quad \exists \sigma : \sigma \models_{\mathcal{H}^*} \{a \leq a | a \in M\} \quad \text{(denn jede Substitution erfüllt } a \leq a) \\
&\Longrightarrow \quad \exists \sigma : \sigma \models_{\mathcal{H}^*} C^0
\end{aligned}$$

Induktionsschritt $n \to n + 1$:
Wir betrachten $C^{n+1} = C \circ C^n$. Nach der Definition von $\circ$ gilt: $a_1 \leq a_2 \in C^{n+1}$, falls

$$\exists a \in M : a_1 \leq a \in C^n \text{ und } a \leq a_2 \in C^n$$

Nach Induktionsvoraussetzung existiert eine Substitution σ mit $\sigma \models_{\mathcal{H}^*} C^n$:

$$\begin{aligned}
\sigma \models_{\mathcal{H}^*} a_1 \leq a \text{ und } \sigma \models_{\mathcal{H}^*} a \leq a_2 \quad &\Longrightarrow \quad (\sigma a_1, \sigma a) \in \mathcal{H}^* \text{ und } (\sigma a, \sigma a_2) \in \mathcal{H}^* \quad \text{(nach Def. 5.3.8)} \\
&\Longrightarrow \quad (\sigma a_1, \sigma a_2) \in \mathcal{H}^* \quad \text{(da } \mathcal{H}^* \text{ transitiv)} \\
&\Longrightarrow \quad \sigma \models_{\mathcal{H}^*} a_1 \leq a_2
\end{aligned}$$

Damit ist gezeigt, dass $\models_{\mathcal{H}^*} a_1 \leq a_2$ für alle $a_1 \leq a_2 \in C^{n+1}$ und damit auch $\models_{\mathcal{H}^*} C^{n+1}$.

Als zusätzliche Voraussetzung für den Beweis von Satz 5.6.1 verlangen wir, dass alle in einem Programm definierten Templates einen gemeinsamen Vorfahren T besitzen. T besitzt keine Methoden und keine Attribute. Für die Praxis bedeutet diese Voraussetzung weder eine Einschränkung noch einen Gewinn: Es ist mit einem solchen Vorfahren T zwar beispielsweise möglich, Objekte verschiedener Klassen in einer Liste aufzusammeln, bei denen dies vorher nicht erlaubt war. Es gibt aber keine Nachricht, die an die Elemente einer solchen Liste geschickt werden können, da in T keine Methoden definiert sind.

Kommen wir nun zum Beweis von Satz 5.6.1. Wir beginnen mit $\Longrightarrow$. Für eine beliebige Contraintmenge C und eine beliebige Vererbungshierachie $\mathcal{H}^*$ ist demnach zu zeigen:

$$\models_{\mathcal{H}^*} C \Longrightarrow \mathcal{H}^* \vdash_5 C^+$$

Wir starten mit $\models_{\mathcal{H}^*} C$. Nach Satz 5.6.2 gilt dann $\models_{\mathcal{H}^*} C^*$ und damit auch $\models_{\mathcal{H}^*} C^+$. Daraus folgt nach Definition 5.3.8

$$\exists \sigma : \forall c \in C^+ : \sigma \models_{\mathcal{H}^*} c$$

Mit Hilfe dieser Voraussetzung werden wir zeigen, dass $\mathcal{H}^* \vdash_5 C^+$. Dazu wenden wir die Regeln aus Tabelle 5.9 rückwärts an: Aus C^+ lassen sich auf diese Weise alle Elemente genau dann entfernen, wenn sich umgekehrt aus der leeren Menge die Menge C^+ konstruieren lässt.

1. Constraints der Form $a \leq b$ lassen sich mit Hilfe der Regel [Variablen] entfernen. Enthält die Constraintmenge keine Constraints dieser Form mehr, fahren wir mit der nächsten Regel fort.

2. Constraints der Form $A \leq a$ lassen sich ohne zusätzliche Bedingung über die Regel [Oberklasse] entfernen.

3. Enthält C mehrere Constraints der Form $a \leq X$, wobei a eine feste Variable und X ein beliebiger Templatetyp ist, lassen sich alle bis auf einen mit Hilfe der Regel [Unterklasse$_2$] entfernen. Gehen wir also davon aus, dass C mindestens zwei Constraints $a \leq A$ und $a \leq B$ enthält. Nach Voraussetzung existiert eine Substitution σ, die C^+ erfüllt. Daher erfüllt sie auch die Constraints $a \leq A$ und $a \leq B$. Es sei

   $$\sigma(a) = D$$

   Dann gilt

   $$(D, A) \in \mathcal{H}^* \text{ und } (D, B) \in \mathcal{H}^*$$

   Da wir keine Mehrfachvererbung zulassen, gilt

   $$D = A \text{ oder } D = B$$

   Also gilt

   $$(A, B) \in \mathcal{H}^* \text{ oder } (B, A) \in \mathcal{H}^*$$

   Durch Anwendung von Regel [Unterklasse$_2$] lässt sich also entweder $a \leq A$ oder $a \leq B$ aus C entfernen. Auch dieser Schritt wird sooft wie möglich angewendet. Zum Schluss enthält die Constraintmenge nur noch maximal einen Constraint der Form $a \leq X$ für jede Typvariable a.

4. Enthält C nur einen Constraint der Form $a \leq A$, so lässt sich dieser durch Anwendung der Regel [Unterklasse$_1$] entfernen.

5. Jeder Constraint der Form $A \leq B$ kann durch Anwendung der Regel [Templates] entfernt werden, da $(A, B) \in \mathcal{H}^*$. Wäre dies nämlich nicht der Fall, so wäre $A \leq B$ nicht erfüllbar und damit auch C nicht erfüllbar.

   Damit haben wir gezeigt, dass C^+ auf die leere Menge zurückgeführt werden kann. Führt man die dabei verrichteten Schritte in der umgekehrten Reihenfolge aus, so konstruiert man C^+. Also gilt $\mathcal{H}^* \vdash_5 C^+$.

Für die umgekehrte Richtung $\Longleftarrow$ ist zu zeigen:

$$\mathcal{H}^* \vdash_5 C^+ \implies \models_{\mathcal{H}^*} C$$

Der Beweis erfolgt durch Induktion über die Regeln aus Tabelle 5.9. Wir starten mit der Regel:

$$[\text{Axiom}] \quad \frac{}{\mathcal{H}^* \vdash_1 \{\}}$$

Die Substitution $\sigma := id$ erfüllt $\{\}$.

Der *Induktionsschritt* erfolgt über alle Regeln der Form

$$\frac{\mathcal{H}^* \vdash_i C_1}{\mathcal{H}^* \vdash_i C_2}$$

Wir müssen zeigen, dass $\models_{\mathcal{H}^*} C_2$ unter der Voraussetzung, dass $\models_{\mathcal{H}^*} C_1$.
Es gilt nach der Definition der Erfüllbarkeit einer Menge von Constraints

$$\models_{\mathcal{H}^*} C_1 \implies \exists \sigma : \forall c \in C_1 : \sigma \models_{\mathcal{H}^*} c$$

und

$$\exists \sigma' : \forall c \in C_2 : \sigma' \models_{\mathcal{H}^*} c \implies \models_{\mathcal{H}^*} C_2$$

Es genügt also zu zeigen, dass eine Substitution σ' existiert, die alle Constraints aus C_2 erfüllt, falls eine Substitution σ existiert, die alle Constraints aus C_1 erfüllt. Wir führen daher den Beweis, indem wir aus einer gegebenen Substitution σ, die C_1 erfüllt, eine Substitution σ' konstruieren und anschließend zeigen, dass $\sigma' \models_{\mathcal{H}^*} C_2$.

- Für die Regel [Übergang] wählen wir $\sigma' := \sigma$. Trivialerweise gilt $\sigma' \models_{\mathcal{H}^*} C$, falls $\sigma \models_{\mathcal{H}^*} C$.

- Da wir die übrigen Regeln nur in der festgelegten Reihenfolge anwenden können, beginnen wir mit der Regel

$$[\text{Templates}] \quad \frac{\mathcal{H}^* \vdash_1 C \quad (A,B) \in \mathcal{H}^*}{\mathcal{H}^* \vdash_1 \{A \leq B\} \cup C}$$

Nach Induktionsvoraussetzung existiert eine Substitution σ mit $\sigma \models_{\mathcal{H}^*} C$. Wir setzen $\sigma' := \sigma$. Es gilt $\sigma' \models_{\mathcal{H}^*} \{A \leq B\} \cup C$, denn jede Substitution erfüllt trivialerweise $\{A \leq B\}$.

- Als nächstes betrachten wir die Regel

$$[\text{Unterklasse}_1] \quad \frac{\mathcal{H}^* \vdash_2 C \quad \forall X \in dom(\mathcal{H}^*) : \ (a \leq X) \notin C}{\mathcal{H}^* \vdash_2 \{a \leq A\} \cup C}$$

Wir setzen

$$\sigma' := \sigma[a/A] \tag{5.1}$$

Offensichtlich gilt

$$\sigma' \models_{\mathcal{H}^*} \{a \leq A\} \tag{5.2}$$

Nach Voraussetzung der Regel [Unterklasse$_1$] gilt:

$$\forall X \in dom(\mathcal{H}^*) : \ (a \leq X) \notin C$$

Da zudem bisher nur die Regeln [Axiom] und [Templates] angewendet wurden, enthält C nur Constraints der Formen $A \leq B$ und $b \leq X$ wobei $b \neq a$. Daher gilt

$$\sigma' \models_{\mathcal{H}^*} C \tag{5.3}$$

Aus 5.2 und 5.3 folgt:

$$\sigma' \models_{\mathcal{H}^*} \{a \leq A\} \cup C$$

- Es folgt die Regel

$$[\text{Unterklasse}_2] \quad \frac{\mathcal{H}^* \vdash_3 \{a \leq B\} \cup C \quad (B,A) \in \mathcal{H}^*}{\mathcal{H}^* \vdash_3 \{a \leq A\} \cup \{a \leq B\} \cup C} \tag{5.4}$$

Wir setzen $\sigma' := \sigma$. Sei $\sigma(a) = \sigma'(a) = Z$. Dann gilt:

$$(Z,B) \in \mathcal{H}^* \quad (\text{da } \sigma' \models_{\mathcal{H}^*} \{a \leq B\}) \tag{5.5}$$

Ferner gilt:

$$(B,A) \in \mathcal{H}^* \quad (\text{nach Voraussetzung in 5.4}) \tag{5.6}$$

Da $\mathcal{H}^*$ transitiv ist, folgt aus 5.5 und 5.6 $(Z,A) \in \mathcal{H}^*$. Also gilt nach Definition der Erfüllbarkeit eines Constraints $\sigma' \models_{\mathcal{H}^*} \{a \leq A\}$. Da $\sigma' = \sigma$ gilt damit $\sigma' \models_{\mathcal{H}^*} C$ und damit auch: $\sigma' \models_{\mathcal{H}^*} \{a \leq A\} \cup \{b \leq A\} \cup C$.

- Betrachten wir nun die Regel

$$[\text{Oberklasse}] \quad \frac{\mathcal{H}^* \vdash_4 C}{\mathcal{H}^* \vdash_4 \{A \leq a\} \cup C}$$

Nun machen wir eine Fallunterscheidung über $\sigma(a)$:

$\sigma(a) = a$	$\sigma(a)$ wurde bislang noch nicht verändert. Also enthält kein Constraint aus C die Typvariable a. Wir setzen $\sigma' := \sigma[a/T]$, wobei T der gemeinsame Vorfahre aller Templates ist. Offensichtlich gilt deshalb, dass $\sigma' \models \{A \leq a\}$. Außerdem gilt $\sigma' \models C$, da kein Constraint von C die Variable a enthält. Also gilt $\sigma' \models \{A \leq a\} \cup C$.
$\sigma(a) = T$	Dieser Fall tritt auf, wenn C bereits einen weiteren Constraint der Form $a \leq X$ enthielt (X beliebiger Templatetyp). Wir wählen $\sigma' := \sigma$. Da T der Vorfahre aller Templates ist, gilt $\sigma' \models \{A \leq a\}$. Trivialerweise gilt auch $\sigma' \models C$. Also gilt $\sigma' \models \{A \leq a\} \cup C$.
$\sigma(a) = X$, $X \neq T$	Dieser Fall kann nur auftreten, wenn $a \leq X \in C$, denn Gleichung 5.1 ist die einzige Stelle, an der $\sigma(a)$ auf einen Wert ungleich T gesetzt wird. Wir wählen $\sigma' := \sigma$. Als Endergebnis der gesamten Berechnung möchten wir eine transitive Relation erhalten. Aus diesem Grund muss C auch den Constraint $A \leq X$ enthalten, da Constraints dieser Form nur durch die bereits abgearbeitete Regel [Templates] gewonnen werden können. Zum Einfügen des Constraints $A \leq X$ musste jedoch die Bedingung $(A, X) \in \mathcal{H}^*$ erfüllt gewesen sein. Also gilt $\sigma' \models \{A \leq a\}$. Trivialerweise gilt $\sigma' \models C$. Also gilt $\sigma' \models \{A \leq a\} \cup C$.

- Als letztes fehlt noch die Regel:

$$[\text{Variablen}] \quad \frac{\mathcal{H}^* \vdash_5 C}{\mathcal{H}^* \vdash_5 \{a \leq b\} \cup C}$$

Auch hier müssen wir eine Fallunterscheidung über σ durchführen:

$\sigma(a) = A$, $\sigma(b) = B$	Wir wählen $\sigma' := \sigma$. Da Gleichung 5.1 die einzige Stelle ist, an der $\sigma(b)$ auf den Wert B gesetzt werden kann, muss $b \leq B \in C$ sein. Damit das Endergebnis der gesamten Berechnung transitiv sein kann, muss deshalb auch $a \leq B \in C$ sein. Daraus folgt, dass $A \leq B \in C$. Also ist klar, dass $\sigma' \models_{\mathcal{H}^*} \{a \leq b\}$. Daher gilt $\sigma' \models \{a \leq b\} \cup C$.
$\sigma(a) = A$, $\sigma(b) = A$	Wir wählen $\sigma' := \sigma$. Trivialerweise gilt $\sigma' \models \{a \leq b\} \cup C$.
$\sigma(a) = A$, $\sigma(b) = b$	Wir wählen $\sigma' := \sigma[b/A]$. Trivialerweise gilt $\sigma' \models \{a \leq b\}$. Da kein Constraint aus C die Variable b enthält, gilt auch $\sigma' \models C$ und damit auch $\sigma' \models \{a \leq b\} \cup C$.

$\sigma(a) = a$, $\sigma(b) = B$	Wir wählen $\sigma' := \sigma[a/B]$. Trivialerweise gilt $\sigma' \models \{a \leq b\}$. Da kein Constraint aus C die Variable a enthält, gilt auch $\sigma' \models C$ und damit auch $\sigma' \models \{a \leq b\} \cup C$.
$\sigma(a) = a$, $\sigma(b) = b$	In diesem Fall kommen weder die Variable a noch die Variable b in einem Constraint aus C vor. Daher können wir willkürlich $\sigma' := \sigma[a/T, b/T]$ wählen. Es ist klar, dass $\sigma' \models \{a \leq b\}$. Da weder a noch b in einem Constraint aus C vorkommen, gilt auch $\sigma' \models C$ und damit auch $\sigma' \models \{a \leq b\} \cup C$.

Damit wäre gezeigt, dass eine Menge von Constraints genau dann erfüllt ist, wenn syntaktisch durch die Regeln aus Tabelle 5.9 die transitive Hülle dieser Menge erzeugt werden kann.

6. Implementierung

Bestandteil der vorliegenden Arbeit ist ein **Compiler**, der ein Programm aus ObjectCurry in Curry übersetzt, eine zusätzliche Typprüfung vornimmt und die fehlenden Typen inferiert. Der Compiler selbst ist komplett in Curry geschrieben. Er besteht aus Scanner, Parser, Codegenerator und Typchecker. Diese vier Bestandteile liegen jeweils in Form eines eigenen Moduls vor.

Der Übersetzungsprozess läuft in mehreren Phasen ab. Zunächst wird die Datei, die das Quellprogramm enthält, eingelesen und vom Scanner in eine Folge von Token übersetzt. Der anschließende Parseprozess erzeugt den abstrakten Syntaxbaum des Programms und gibt diesen an Typchecker und Codegenerator weiter. Der Typchecker inferiert die fehlenden Typen und prüft auf Grundlage des in Kapitel 5 vorgestellten Typsystems das Programm auf Typfehler. Ist das Programm korrekt, so fließen die im Typchecker durch Typinferenz gewonnenen Informationen zusätzlich neben dem abstrakten Syntaxbaum als Eingabe in den Codegenerator. Dieser erzeugt ein Curry-Programm, das anschließend mit Hilfe des PAKCS [H$^+$00] ausgeführt werden kann.

Die folgenden Abschnitte geben einen kurzen Überblick über den Aufbau und die Besonderheiten von Scanner, Parser, Codegenerator und Typchecker. Im Detail beschreiben wir den Parser, da er eine spezielle Technik zur Behandlung der Offside-Rule verwendet.

6.1. Scanner

Der **Scanner** bildet die erste Phase des Compilers. Seine Aufgabe besteht darin, den Quelltext zeichenweise einzulesen und eine Folge von Symbolen (engl. *token*) zurückzuliefern, die den Ausgangspunkt der anschließenden Syntaxanalyse darstellt [ASU90]. Der vorliegende Scanner erzeugt spezielle Symbole für Bezeichner, Operatoren, Integerzahlen[1], Strings, einzelne Character, Operatoren, öffnende und schließende Klammern, sowie die in Curry und ObjectCurry reservierten Schlüsselworte.

Im Compilerbau wird ein Scanner meist in Form eines endlichen Automaten definiert. Wir weichen von diesem Ansatz jedoch ab, um dem Scanner zusätzliche Aufgaben übertragen zu können. Einem endlichen Automaten ist es beispielsweise nicht möglich, geschachtelte Kommentare zu erkennen. Die Syntax von Curry lässt diese aber zu. Wir haben uns entschieden, Kommentare bereits im Scanner zu erkennen, um den Parser von dieser Aufgabe zu entlasten.

[1] Fließkommazahlen sind nicht implementiert, da diese auch in der aktuellen Version des PAKCS bislang fehlen

Bezeichner und Schlüsselworte bestehen aus einem Buchstaben, gefolgt von einer beliebigen Kombination aus Buchstaben, Ziffern und Unterstrichen. Zusätzlich können **Bezeichner von Modulen** Punkte enthalten. Der Punkt bereitet ein zusätzliches Problem beim Bau von Scanner und Parser für Curry, denn er besitzt zwei Funktionen: Zum einen wird er als Operator für die Sequenz zweier Funktionen verwendet, wie z. B. in `f.g`. Zum anderen dient er als Trennsymbol zwischen Modulnamen und Bezeichnern, z. B. in `List.foldl`. Das erste Vorkommen eines Modulbezeichners innerhalb eines Moduls tritt immer in Form eines `import`- oder `module`-Konstrukts auf. Außerdem muss auf das Schlüsselwort `import` bzw. `module` stets ein Modulbezeichner folgen. Diese Tatsache ermöglicht es, den Scanner so zu gestalten, dass er Modulnamen als spezielle Token liefert: Hat der Scanner das Schlüsselwort `module` bzw. `import` erkannt, so liest er den nächsten Bezeichner inklusive aller Punkte, fügt diesen als Modulbezeichner-Token in die Ausgabeliste ein, und vermerkt zusätzlich in einer internen Liste diesen Bezeichner als eingeführten Modulbezeichner. Sobald der Scanner im weiteren Verlauf auf einen Eintrag dieser Liste trifft, gibt er den Bezeichner als Modulbezeichner-Token zurück. Alle anderen Bezeichner werden, sofern es sich nicht um Schlüsselworte handelt, als normale Bezeichner-Token ohne Punkte zurückgegeben.

Infixoperatoren sind beliebige Kombinationen der Zeichen „`~!@#$%^&*+-=<>?./|\:`“. Für einige Zeichenfolgen wie „`|`“ oder „`:=`“ wurden spezielle Token definiert, um Syntaxfehler früher erkennen zu können.

Curry bzw. ObjectCurry verfügen über die folgenden reservierten Schlüsselworte, die nicht als Bezeichner verwendet werden können:

Schlüsselworte in Curry:

`as`	`choice`	`data`	`else`	`external`	`flex`	`free`
`hiding`	`if`	`import`	`in`	`infix`	`infixl`	`infixr`
`let`	`modules`	`pragma`	`qualified`	`rigid`	`then`	`where`

zusätzliche Schlüsselworte in ObjectCurry:

`template`	`constructor`	`methods`	`extends`	`super`	`delay`	`continue`

Die Implementierung des Scanners liegt in Form eines Moduls mit dem Namen `scanner.curry` vor. Die Funktion `scanner` liest einen String ein und gibt eine Folge von Token zusammen mit der Position dieses Token im String zurück.

```
data TokenAndPos = TokenAndPos Token Int Int
scanner :: String -> [TokenAndPos]
```

6.2. Parser

Die zweite Phase des Compilers ist der **Parser**. Seine Aufgabe ist es, aus der Liste von Token, die der Scanner liefert, den abstrakten Syntaxbaum des Programms aufzubauen.

Der vorliegende Parser kann sowohl zum Parsen von Curry- als auch von ObjectCurry-Programmen verwendet werden. Er eignet sich damit auch zur Erstellung künftiger Erweiterungen von Curry.

Implementiert wurde er durch die Verwendung von Parserkombinatoren. **Parserkombinatoren** erlauben den einfachen Aufbau eines Parsers entlang der kontextfreien Grammatik der zu parsenden Sprache. Es ist lediglich notwendig, Linksrekursion aus dieser Grammatik zu eliminieren.

Caballero und L'opez-Fraguas [CLF99] zeigen, wie in funktional-logischen Programmiersprachen Parserkombinatoren erstellt werden können, die die Vorzüge beider Paradigmen zu nutzen wissen. Sie verwenden dazu die Programmiersprache $\mathcal{TOY}$ [LFSH99]. Eine Übersetzung der in [CLF99] vorgestellten Parser und Kombinatoren in Curry liegt dem PAKCS als Bibliothek bei.

Da Curry und insbesondere ObjectCurry Gebrauch von der **Offside-Rule** machen, ist es jedoch nötig, die Parserkombinatoren entsprechend zu modifizieren.

Zunächst stellen wir die Parserkombinatoren nach [CLF99] in ihrer Übersetzung nach Curry vor. Anschließend werden wir sehen, wie diese durch Einführung neuer Regeln zur Behandlung der Offside-Rule erweitert werden können.

6.2.1. Nicht-deterministische Parserkombinatoren

Ein Parser soll erkennen, ob ein Programm Element einer bestimmten Sprache ist oder nicht. Wir wollen den Parser mit Hilfe von Teilparsern an Hand der EBNF-Regeln, die die Syntax der zu parsenden Sprache beschreiben, sukzessive aufbauen.

Dazu definieren wir einen Parser generell als eine Funktion, die eine Liste von Token entgegen nimmt, einige Token verarbeitet, und die restliche Liste zurückgibt:

```
type Parser token = [token] -> [token]
```

Ein Teilparser zur Erkennung eines einzelnen Terminalsymbols prüft, ob das erste Element der zu parsenden Tokenliste dem Terminalsymbol entspricht und liefert die restlichen Token zurück:

```
terminal :: token -> Parser token
terminal sym (token:tokens) | sym=:=token = tokens
```

Mit Hilfe von Kombinatoren können Teilparser wie beispielsweise `terminal sym` kombiniert werden. Der Aufbau der EBNF-Regeln verlangt zunächst Kombinatoren zur Erkennung von Alternativen und Sequenzen. Die **Alternative** zweier Parser gestaltet sich in funktional-logischen Programmiersprachen wie Curry oder $\mathcal{TOY}$ sehr einfach, da diese die Definition von nicht-deterministischen Funktionen erlauben (s. 1.2.2). Wir definieren einen Operator `<|>`, der die Kombination zweier alternativer Parser erlaubt: Der Ausdruck `p <|> q` ist ein Parser, der sich entweder wie der Parser `p` oder der Parser `q` verhält. In Curry lässt sich `<|>` wie folgt formulieren:

```
(<|>) :: Parser token -> Parser token -> Parser token
p <|> _ = \sentence -> p sentence
_ <|> q = \sentence -> q sentence
```

Wir nutzen hierbei die Möglichkeit von Curry aus, nicht-deterministische Funktionen zu definieren.
Die **Konkatenation** `<*>` ist komplizierter, durch die Verwendung des Gleichheitsconstraints aber dennoch einfach zu formulieren: Der Parser `p1 <*> p2` ist eine Funktion `seq`, welche eine Folge von Token entgegennimmt. Diese wird zunächst an den Parser `p1` übergeben, und dessen Ergebnis an den Parser `p2`.

```
p1 <*> p2 = seq
 where seq sentence | p1 sentence =:= sent1 = p2 sent1
   where sent1 free
```

Bislang wurden ausschließlich Parser betrachtet, die eine bestimmte Tokenfolge erkennen. In der Regel ist man aber nicht nur daran interessiert, ob ein Programm Element einer bestimmten Sprache ist, sondern parallel zum Parsevorgang soll der abstrakte Syntaxbaum konstruiert werden. Realisieren lässt sich dies durch die Verwendung von Parsern mit Repräsentation. Ein Parser mit Repräsentation erhält neben der Tokenliste die Repräsentation des zu parsenden Ausdrucks als zusätzliches Argument. Die Repräsentation eines Ausdrucks kann z. B. ein Teil des abstrakten Syntaxbaums sein. Beim Aufruf wird eine freie Variable als Repräsentation an den Parser übergeben, der diese an das Ergebnis bindet.

```
type ParserRep rep token = rep -> Parser token
```

Als Beispiel definieren wir einen Parser, der einen beliebigen Operator erkennt und als seine eigene Repräsentation zurückliefert:

```
operator :: ParserRep String Token
operator rep (token:tokens)
  | token =:= (Operator op)
  & rep =:= op
  = tokens
```

Der Operator `>>>` gestattet es, einen Parser ohne Repräsentation zu einem Parser mit Repräsentation zu erweitern:

```
(>>>) :: Parser token -> rep -> ParserRep rep token
parser >>> repexp = attach
 where attach rep sentence | parser sentence =:= rest &> repexp =:= rep
                           = rest
                            where rest free
```

Die Sequenz zweier Parser mit Repräsentation wird mit Hilfe des bereits vorgestellten Operators `<*>` realisiert. Allein für die Alternative ist es nötig, einen neuen Kombinator `<||>` einzuführen:

```
(<||>)  :: ParserRep r t -> ParserRep r t -> ParserRep r t
p <||> q = \rep -> p rep <|> q rep
```

Die vorgestellten Operatoren `<*>`, `<||>` und `>>>` ermöglichen eine komfortable Schreibweise beim Aufbau von Parsern. Als Beispiel betrachten wir die Umsetzung der folgenden EBNF-Regel der Curry-Grammatik:

$$Literal ::= Int|Char|String$$

Zunächst definieren wir einen Datentyp `TLiteral`, der ein Literal repräsentiert.

```
data TLiteral = CLiteral_Int Int | CLiteral_Char String | CLiteral_String String
```

Nun definieren wir drei getrennte Parser zum Erkennen eines Integer-Token, eines Character-Token oder eines String-Token.

```
litInt :: ParserRep TLiteral Token
litInt    = token (NumberInt i)   >>> CLiteral_Int i    where i free

litChar :: ParserRep TLiteral Token
litChar   = token (TokenChar c)   >>> CLiteral_Char c   where c free

litString :: ParserRep TLiteral Token
litString = token (TokenString s) >>> CLiteral_String s where s free
```

Der Parser zum Erkennen eines Literals ist die Alternative dieser drei Parser:

```
literal :: ParserRep TLiteral
literal = litInt <||> litChar <||> litString
```

Die Definition von `literal` zeigt die Ähnlichkeit des Aufbaus eines Parsers mit der entsprechenden EBNF-Regel.

6.2.2. Behandlung der Offside-Rule

Die Syntax von Curry und ObjectCurry macht Gebrauch von der **Offside-Rule**, auch Abseitsregel genannt. Diese definiert Thiemann [Thi94, S. 22] wie folgt:

„Das erste Symbol, was einem der Schlüsselworte `let`, `where` oder `of` folgt, legt den linken Rand des gerade eröffneten **Geltungsbereichs** fest. Für die darauf folgenden Textzeilen gilt:	*get*
Beginnt eine Zeile rechts vom Rand [..], so wird sie als Fortsetzung der darüber liegenden Zeile betrachtet.	*greater*
Beginnt eine Zeile genau am Rand [..], so beginnt mit ihr eine neue Deklaration innerhalb des Geltungsbereichs.	*same*
Beginnt eine Zeile links vom Rand, so liegt sie im Abseits und der Geltungsbereich endet vor der Zeile. Sie (und alle folgenden Zeilen) gehören zu einem umschließenden Geltungsbereich.“	*lower*

Die bisher vorgestellten Parser und Kombinatoren sollen nun um die **Offside-Rule** erweitert werden. Grundsätzlich bieten sich hierzu zwei Alternativen:

1. Durch eine Modifikation des Scanners oder eine zusätzliche Phase zwischen Scanner und Parser werden zusätzliche Token in die Eingabeliste des Parsers eingefügt. Diese repräsentieren die Klammerung, welche sich aus der Anwendung der Offside-Rule ergibt. Der Nachteil bei dieser Vorgehensweise besteht darin, dass die Eingabe des Parsers Token enthält, die nicht im Quelltext stehen. Die Behandlung von Fehlern wird dadurch komplizierter.

2. Jedes Token wird mit einer Positionsangabe versehen, die besagt, in welcher Spalte das Token im Quelltext steht. In der Praxis ist dies ohnehin sehr häufig anzutreffen, da Fehlermeldungen des Parsers auch die Position, an der ein Fehler auftrat, beinhalten soll. Die Beachtung der Offside-Rule wird ganz dem Parser überlassen.

Die vorliegende Implementierung macht von der zweiten Methode Gebrauch. Der Scanner liefert bereits eine Liste von Token mit Positionsangabe des Typs `[TokenAndPos]`. Diese dient dem Parser als Eingabe.

Ein Parser mit Behandlung der Offside-Rule erhält gegenüber einem der oben vorgestellten Parser zwei zusätzliche Parameter: Der erste beinhaltet die Spalte des aktuellen Geltungsbereichs und der zweite eine Bedingung, die das nächste zu lesende Token erfüllen muss. Der Typ eines Parsers hat daher folgendes Aussehen:

```
type OffsideRuleParser = Int -> Condition -> Parser TokenAndPos
```

Die Bedingung vergleicht die Spalte des aktuellen Geltungsbereichs mit der Spalte des nächsten Token. Sie besitzt daher zwei Argumente vom Typ `Int`. Wir benötigen zunächst zwei Bedingungen: Soll das nächste Token innerhalb derselben Deklaration im selben Geltungsbereich liegen, so müssen wir sicherstellen, dass die Spalte des Token größer als die des aktuellen Geltungsbereichs ist (*greater*). Beginnt mit dem nächsten Token eine neue Deklaration, so ist zu überprüfen, ob die Spalte des Token und die des Geltungsbereichs übereinstimmen. Des Weiteren kann ein Token einen neuen Geltungsbereich einleiten. Dann bestimmt dieses die Position des nächsten Geltungsbereichs. Zu diesem Zweck definieren wir eine dritte Bedingung (*get*).

```
type Condition = Int -> Int -> Success

greater :: Condition
greater c1 c2 = (c2>c1)=:=True

same :: Condition
same c1 c2 = (c2==c1)=:=True

get :: Condition
get c1 c2 = c1=:=c2
```

Die eigentliche Prüfung der Bedingung übernimmt der Parser zur Erkennung von Terminalsymbolen, denn nur dieser kennt die Position des nächsten Token.

```
terminal sym c1 condition ((TokenAndPos token c2 _ ):tokens)
  | (condition c1 c2) & sym=:=token = tokens
```

Für die Konkatenation definieren wir nun drei neue Operatoren. Gehört das nächste Token zur gleichen Deklaration im aktuellen Geltungsbereich, so verwenden wir den Operator `<*:>`. `<*:>` unterscheidet sich gegenüber `<*>` dadurch, dass er an den zweiten Teilparser die Bedingung `greater` übergibt, die sicherstellt, dass das nächste Token tatsächlich in der gleichen Deklaration liegt. Dazu wird überprüft, ob die Spalte des nächsten Token größer ist als die des aktuellen Geltungsbereichs.

```
(<*:>) :: OffsideRuleParser -> OffsideRuleParser -> OffsideRuleParser
p1 <*:> p2 = seq
 where seq c ind sentence | p1 c ind sentence =:= sent1 = p2 c greater sent1
  where sent1 free
```

Beginnt jedoch mit dem nächsten Token eine neue Deklaration, so setzen wir den Operator `<\/>` ein. Dieser unterscheidet sich von `<*:>` nur durch die Verwendung der Bedingung `same` anstelle von `greater`.

```
<\/> :: OffsideRuleParser -> OffsideRuleParser -> OffsideRuleParser
p1 <\/> p2 = seq
 where seq c ind sentence | p1 c ind sentence =:= sent1 = p2 c same sent1
  where sent1 free
```

Als dritte Möglichkeit kann ein Token einen neuen Geltungsbereich eröffnen. In diesem Fall verwenden wir die Bedingung `get`, die die Spalte des Geltungsbereichs an die Spalte des nächsten Token bindet. Als Spalte des Geltungsbereichs übergeben wir daher eine freie Variable.

```
(<*:*>) :: OffsideRuleParser -> OffsideRuleParser -> OffsideRuleParser
p1 <*:*> p2 = seq
 where seq c ind sentence | p1 c ind sentence =:= sent1 = p2 nc get sent1
  where sent1,nc free
```

Ein OffsideRule-Parser mit Repräsentation besitzt den folgenden Typ:

```
type OffsideRuleParserRep rep = rep -> OffsideRuleParser
```

Die Definition des Operators `<||>` für die Alternative kann unverändert übernommen werden, lediglich der Typ muss entsprechend geändert werden.

```
(<||>)  :: OffsideRuleParserRep rep -> OffsideRuleParserRep rep
            -> OffsideRuleParserRep rep
p <||> q = \rep -> p rep <|> q rep
```

Als Beispiel betrachten wir einen Parser, der eine definierende Gleichung einer Curry-Funktion mit einer Liste von lokalen Definitionen nach den folgenden EBNF-Regel erkennt:

$$Equat ::= FunLHS = Expr \; \texttt{where} \; \{localDef, \ldots, localDef\}$$

Die Gleichung besteht aus der linken Seite, dem Zeichen `=`, einem Ausdruck, dem Schlüsselwort `where` und einer Liste von untereinander stehenden Definitionen. Dabei kann es sich sowohl um Funktionsdefinitionen, als auch um Deklarationen freier Variablen halten.

Der Parser `funLHS` erkennt die linke Seite, der Parser `expr` einen Ausdruck. Beide Parser möchten wir nur verwenden, nicht aber ihre Definition angeben. Der Parser `localDef` erkennt eine einzelne Funktionsdefinition oder die Deklaration einer freien Variable.

```
data Equation = Equation FunLHS Expr [ValDecl]

equat =      funLHS f <*:> token Equal <*:> expr e <*:> token Where
             <*:*> (among localDef) vds >>> Equation f e vds
  where f,e,vds free
```

Zum Verknüpfen von `funLHS`, `token Equal`, `expr` und `token Where` verwenden wir den Operator `<*:>`, da die von diesen Teilparsern erkannten Tokenfolgen nebeneinander notiert werden müssen. Das Token hinter dem Schlüsselwort `where` öffnet jedoch eine neuen Geltungsbereich. Daher verwenden wir zur Verknüpfung von `token Where` und dem folgenden Parser den Operator `<*:*>`. Auf `where` folgt eine Liste von untereinanderstehenden lokalen Definitionen. Eine einzelne dieser Defintionen erkennt der Parser `valdecl`. Um die gesamte Liste zu erkennen, wenden wir die Funktion `among` auf den Parser `valdecl` an: `among valdecl` erkennt eine Liste von untereinanderstehenden Definitionen.

```
among :: ParserRep rep -> ParserRep [rep]
among parser =
       empty >>> []
  <||> ((parser a) <\/> ((among parser) as)) >>> (a:as)
  where a,as free
```

`empty` ist dabei der Parser, der das leere Wort erkennt:

```
empty :: Parser
empty _ _ seq = seq
```

6.2.3. Gesamtaufbau und Verwendung von Parsern

Der gesamte ObjectCurry-Parser ist aus einer Vielzahl von Einzelparsern aufgebaut. Jeder Einzelparser besitzt den Typ `OffsideRuleParserRep rep`. `rep` ist dabei jeweils ein abstrakter Datentyp zur Darstellung eines Teils des abstrakten Syntaxbaums. Der Parser `expr` beispielsweise erkennt einen beliebigen Ausdruck und ist vom Typ `OffsideRuleParserRep TExpr`, wobei `TExpr` ein abstrakter Datentyp zur Darstellung eines Ausdrucks ist. Der Parser zum Erkennen eines kompletten ObjectCurry-Moduls trägt den Namen `curryModule` und besitzt den Typ `OffsideRuleParserRep TCurryModule`.

Wie wir wissen ist `OffsideRuleParserRep TExpr` ein Typsynonym für

```
TCurryModule -> Int -> Condition -> [TokenAndPos] -> [TokenAndPos]
```

Das Parsen eines kompletten ObjectCurry-Programms geschieht über einen Ausdruck der Form

```
curryModule ast 0 get (scanner program)
```

wobei program ein String ist, der das zu parsende Programm enthält und ast eine freie logische Variable. Die Funktion curryModule ist nicht-deterministisch. Ist das Ergebnis die leere Liste [], d. h. die Resttokenliste ist leer, so enthält ast den durch den Parser erzeugten abstrakten Syntaxbaum des Programms. Daher unifzieren wir den Ausdruck mit der leeren Liste durch Verwendung des Gleichheitsconstraints. Um den Parser innerhalb der I/O-Monade verwenden zu können, machen wir ihn mit Hilfe der **eingekapselten Suche** deterministisch, indem wir nur die erste Lösung betrachten. Dies ist möglich, da die dem Parser zugrunde liegende Grammatik eindeutig ist.

```
parser :: parser -> tokens -> TCurryModule
parse tokens
  = findfirst (\ast -> parser ast 0 get tokens =:= [])
```

Der Ausdruck parse curryModule (scanner program) liefert den abstrakten Syntaxbaum des Programms, das der String program enthält.
Zu Testzwecken wurde zu jedem Teilparser p zusätzlich eine Funktion mit dem Namen code_p implementiert, die aus dem abstrakten Syntaxbaum oder einem Teil davon wieder den ursprünglichen String konstruiert.

Bsp.: Die Funktion

```
code_curryModule.(parse curryModule).scanner :: String -> String
```

erhält ein ObjectCurry-Programm als Argument, scannt und parst dieses, und erzeugt aus dem dabei erzeugten abstrakten Syntaxbaum wieder eine Stringdarstellung des Ausgangsprogramms.

6.2.4. Fehlerbehandlung

Ein syntaktischer Fehler liegt genau dann vor, wenn der Parser nicht die gesamte Tokenliste verarbeiten kann, da er auf ein Token trifft, auf dass keine der angegebenen Regeln angewendet werden kann. Unifizieren wir die Anwendung einer Parser-Funktion mit einer freien Variable l anstelle der leeren Liste, so wird l an eine Liste mit den Token gebunden, die nicht geparst werden konnten:

```
curryModule ast 0 get (scanner program) =:= l
```

Durch den Nichtdeterminismus des Parsers erhalten wir eine Menge von Lösungen dieses Constraints. Wir interessieren uns jedoch nur für diejenige mit der kürzesten Restliste l. Das erste Element von l ist in diesem Fall das erste Token, das nicht mehr geparst werden konnte; zusammen mit der Positionsangabe kann eine Fehlermeldung ausgegeben werden, die dem Benutzer einen konkreten Hinweis auf den Syntaxfehler im Programm liefert.

Die Ermittlung der Lösung mit kleinster Resttokenlisten läuft letztendlich auf die Ermittlung aller Lösungen heraus, von denen anschließend diejenige mit der kleinste Resttokenliste ausgewählt wird.

In der vorliegenden Implementierung nützen wir eine spezielle Eigenschaft das PAKCS aus. Der Compiler curry2prolog ermittelt die Lösungen über Backtracking. Die Reihenfolge der definierenden Gleichung hat Einfluss auf die Lösung, die dabei zuerst berechnet wird. Im Parser wurden die Gleichungen jedes Teilparsers daher so angeordnet, dass zunächst diejenigen Regeln angewendet werden, die die meisten Token lesen. Daher ist die erste Lösung immer diejenige mit der kleinsten Resttokenliste, so dass die aufwendige Berechnung und der Vergleich aller Lösungen entfällt.

Nicht implementierte Funktionen

Die vorliegende Implementierung erkennt keine Ausdrücke mit Infixoperatoren, bei denen die Auflösung der Assoziativität erforderlich ist. Ein Ausdruck wie `3+4+5` oder `3+4*5` muss daher zu `(3+4)+5` bzw. `3+(4*5)` geklammert werden. Die Angabe von Präzedenzen und Assoziativität der Infixoperatoren über `infix`, `infixl` bzw. `infixr` werden momentan ignoriert.

6.3. Codegenerierung

Die Codegenerierung erfolgt in dem Modul `builder.curry`. Als Eingabe dient der vom Parser erzeugte abstrakte Syntaxbaum. Erzeugt wird ein Curry-Programm mit der gleichen Syntax wie das ursprüngliche ObjectCurry-Programm. Herkömmliche Funktions- und Typdefinitionen können unverändert in die Ausgabe übernommen werden. Dies lässt sich mit Hilfe der im vorigen Abschnitt vorgestellten `code`-Funktionen bewerkstelligen.

Die Übersetzung von Templates erfolgt wie bei den in Kapitel 3 vorgestellten Beispielen. Auf die Implementierung möchten wir an dieser Stelle nicht im Detail eingehen, sondern nur grob die Vorgehensweise darstellen. Zunächst werden die für die Erzeugung des Nachrichtendatentyps und der Funktionen jedes Templates benötigten Informationen in einem seperaten Durchlauf über den abstrakten Syntaxbaum aufgesammelt. Die Namen der dazu verwendeten Funktionen beginnen mit der Zeichenkette `collect`. Anschließend wird daraus der Zielcode erzeugt. Die dafür zuständigen Funktionen tragen Namen, die mit `code` beginnen.

Tabelle 6.1 zeigt die Typen von Objekten, Nachrichten und Methoden auf ObjectCurry-Ebene und in dem vom Compiler erzeugten Curry-Programm. Zur Darstellung von Nachrichten verschiedener Templates wird ein einziger Datentyp namens `Msg` verwendet. Dies ermöglicht Subtyppolymorphismus über Objekten und Nachrichten. Der Typchecker stellt sicher, dass keinem Objekt eine Nachricht gesendet werden kann, das diese nicht versteht.

Die benötigten Typen der Argumente jeder Nachricht werden durch den Typchecker inferiert.

6.3.1. Der Typchecker

Die Implementierung des Typcheckers erfolgte nach dem Vorbild von Mark Jones „Typing Haskell in Haskell“ [Jon99]. Dieser musste modifizert werden, um eingeschränkten Typen nach dem in Kapitel 5.6 vorgestellten Algorithmus zu erhalten. Die genaue Arbeitsweise des Typchecker soll an dieser Stelle nicht beschrieben werden. Stattdessen beschreiben wir

	Typ in ObjectCurry	Typ in Curry
Nachricht	`Message` $a \mid a \leq T$	`Msg`
Objekt	`Object` $a \mid T \leq a$	`Port Msg`
Konstruktor	`Constructor T`	`Port Msg -> [Msg] -> Success`

`new`	`Constructor` a
	`-> Object` b
	`-> Succeess` $\mid a \leq T$
		`Port Msg -> [Msg] -> Success`
		`-> Port Msg`
		`-> Succeess`

`send`	`Message` a
	`-> Object` b
	`-> Success` $\mid b \leq a$
		`Msg`
		`-> Port Msg`
		`-> Succeess`

Tabelle 6.1.: Die Typen von Objekten, Nachrichten und Konstruktoren sowie von `send` und `new` in ObjectCurry und Curry

die wesentlichen Unterschiede zu Jones' Programm.

Als Eingabe für das Programm dient der vom Parser erzeugte abstrakte Syntaxbaum. Im Gegensatz zu Jones arbeiten wir nicht auf einer vereinfachten Core-Sprache, sondern direkt über der gesamten Syntax von ObjectCurry. Dies führt zu einer Reihe weiterer Regeln, die jedoch alle analog zu denen aus „Typing Haskell in Haskell“ aufgebaut sind.

Jones verwendet in seinem Programm eine Zustandsmonade, die es ihm ermöglicht, eine globale Substitution aufzubauen und frische Typvariablen zu produzieren, ohne die dazu nötigen Informationen in jeder Funktion mitführen zu müssen. Curry stellt in der aktuellen Version keine Zustandsmonaden zur Verfügung. Wir verwenden deshalb ein Objekt, das dieselben Aufgaben wie Jones Zustandsmonade übernimmt. Realisiert wurde das Objekt mit der Technik aus Kapitel 2, auf der auch ObjectCurry basiert.

Da Curry keine Typklassen und damit kein Überladen von Funktionen kennt, mussten einige Bezeichner geändert werden.

Subtypconstraints und eingschränkte Typen sind wie folgt definiert:

```
data Constraint = CLe Type Type
type Constraints = [Constraint]
data ConstrainedType = TCT Type Constraints
```

Anstelle von Typschemata verwendet das Programm eingeschränkte Typschemata. Die einzelnen Regeln zum Typen von Ausdrücken und Funktionen wurden dahingehend angepasst, dass sie die entstehenden Constraints wie der Algorithmus aus Kapitel 5.6 aufsammeln. Während des Inferierens der Typen findet keine weitere Überprüfung der Constraints statt.

Eine zusätzliche Funktion wurde für die Behandlung von Templates integriert. Sie inferiert die Typen der Konstruktorfunktion und der im Template definierten Methoden. Dabei verwendet sie eine Attributtypumgebung und eine Vererbungshierachie. Sowohl die Attributtypumgebung als auch die Vererbungshierachie werden in einem seperaten Durchlauf durch den abstrakten Syntaxbaum vor Ablauf des eigentlichen Algorithmus ermittelt.

Nach erfolgreicher Berechnung der eingeschränkten Typen aller in einem Programm definierten Funktionen, werden die darin enthaltenen Constraints überprüft. Der dazu notwendige Algorithmus entspricht exakt dem aus Kapitel 5.6. Die dabei benötigte Bildung der transitiven Hülle aller Constraints übernimmt die Funktion `transitiv`, die durch die Verwendung von Constraints, Nicht-Determinismus und eingekapselter Suche auf sehr einfache Weise implementiert werden kann:

```
isIn x (y:_) = x=:=y
isIn x (_:ys) = isIn x ys

ndt l
  | (isIn (x,y) l)
  = (x,y)
  where x,y free

ndt l
  | (isIn (x,y) l) & (isIn (y,z) l) & ((elem (x,z) l)=:=False)
  = (x,z)
  where x,y,z free

transitiv l =
  let next = findall (\x -> (ndt l) =:= x)
  in if (next==l) then (nub l) else transitiv next
```

6.3.2. Lücken in der Implementierung

Die Implementierung des Typchecker ist nicht vollständig. Daher sind bei der Benutzung folgende Punkte zu beachten:

- Die Reihenfolge bei der Defintion von Funktionen und Templates spielt eine Rolle: Durch Funktions- und Templatedefintionen eingeführte Bezeichner sind nur im nachfolgenden Programm bekannt.
- Die Definition von rekursiven Funktionen ist möglich, nicht jedoch von verschiedenen wechselseitig rekursiven Funktionen.
- Explizite Typangaben werden ignoriert.
- Fehlermeldungen werden zum Teil mit dem eigentlich nur zur Fehlersuche vorgesehenen Constraint `unsafePrint` ausgegeben. Dies kann zu unschönen Effekten in der Ausgabe führen.
- Die Bibliothek `prelude.curry` wird vom Typchecker nicht verarbeitet. Die Typen der wichtigsten Funktionen sind in dem Modul `typing.curry` fest in einer initialen Typumgebung vordefiniert.

Alle Punkte sollten sich ohne größere Probleme nachträglich beseitigen lassen. Auf die Implementierung wurde bislang jedoch verzichtet, da es sich hierbei um Routinearbeiten handelt, die in keinem direkten Zusammenhang mit der Objektorientierung stehen.

7. Zusammenfassung und Ausblick

Mit der vorliegenden Arbeit wurde eine konservative Erweiterung der deklarativen Multiparadigmensprache Curry geschaffen, die zusätzlich zur funktionalen, logischen und nebenläufigen nun auch die objektorientierte Programmierung gestattet.

In Kapitel 1 sind wir zunächst auf die Grundlagen der Objektorientierung sowie der Multiparadigmensprache Curry eingegangen. Die wichtigsten Bestandteile einer objektorientierten Programmiersprache sind Objekte und Klassen: Objekte besitzen einen Zustand und reagieren auf Nachrichten, die sie empfangen. Klassen fungieren als Schablone für die Erzeugung neuer Objekte. Das Konzept der Vererbung zwischen Klassen ermöglicht die Wiederverwendung von Code. Objektorientierte Programmiersprachen verfügen darüber hinaus über Subtyppolymorphismus, der es gestattet, an jeder Stelle, an der ein Objekt einer bestimmten Klasse erwartet wird, auch ein Objekt einer Unterklasse zu verwenden.

Curry verfügt über eine Haskell-ähnliche Syntax und vereint die Konzepte funktionaler, logischer und nebenläufiger Programmiersprachen, wie z. B. verzögerte Auswertung, Funktionen höherer Ordnung, (Gleichheits-)Constraints, logische Variablen und nebenläufige Auswertung von Constraints. Wie ML oder Haskell besitzt auch Curry ein Hindley-Milner-ähnliches Typsystem mit parametrischem Polymorphismus. Ein Vergleich von deklarativen und objektorientierten Programmiersprachen hat die Schwachstellen beider Konzepte gezeigt: Objektorientierte Programmiersprachen bieten in der Regel nur wenig Hilfestellung bei dem Umgang mit Werten im mathematischen Sinne, wie Tupeln, Listen und anderen abstrakten Datentypen. Funktionale Sprachen kennen keine Vererbung als Mittel zur Wiederverwendung von Code und besitzen nur indirekte Möglichkeiten der Kommunikation zwischen verschiedenen Komponenten. Dies hat zu der Überlegung geführt, deklarative Programmiersprachen um Konzepte der Objektorientierung zu erweitern. Curry bietet durch die Verwendung von logischen Variablen oder des integrierten Port-Konzepts zwar weitere Möglichkeiten der Interaktion als rein funktionale Programmiersprachen, bei der Erstellung von graphischen Benutzeroberflächen oder verteilten Anwendung stellen sich aber auch hier Objekte als gewinnbringend heraus. In Abschnitt 1.4 sind daher bereits existierende objektorientierte Erweiterungen funktionaler und logischer Programmiersprachen wie Haskell++, O'Haskell, O'Caml und Vulcan vorgestellt worden.

Motiviert durch diese Überlegungen haben wir in Kapitel 2 eine schon aus [Han99] und [Han00b] bekannte Technik zur objektbasierten Programmierung in Curry vorgestellt. Objekte besitzen in dieser einen gekapselten Zustand und reagieren auf Nachrichten, die an diese gesendet werden. Dargestellt wird ein Objekt durch einen Constraint, also eine Funktion mit dem Rückgabetyp `Success`, die den Zustand des Objekts und einen Strom von Nachrichten empfängt und sich selbst solange rekursiv aufruft, bis der Nachrichtenstrom geschlossen wird. Antworten können durch das Mitsenden freier Variablen innerhalb einer Nachricht übermittelt werden. Das Senden von Nachrichten geschieht durch die schrittweise Instantiierung des Nachrichtenstroms. Wir haben gesehen, dass mehrere Sender einen Merger nötig machen, der jedoch zu weiteren Problemen führt, u. a. zur Verletzung der constant delay property. Der Einsatz von Ports anstelle von Streams löst diese Probleme. Ein Port ist eine mit einem Stream verbundene Multimenge. Stream und Multimenge enthalten dabei stets die gleichen Elemente. Über den Constraint `send` können Elemente in diese Multimenge eingefügt werden.

Diese Vorgehensweise führt in der Praxis zu einem großen Anteil an Code, der rein technischer Natur ist. Dieser erschwert die Lesbarkeit des Programms, birgt auf Grund seiner immer gleichen Struktur ein erhöhtes Risiko an Tippfehlern und verursacht Aufwand bei der Erstellung. Durch eine Erweiterung der Programmiersprache sind dieser Aufwand und die damit verbundenen Nachteile vermeidbar. Eine solche Erweiterung haben wir in Form der Programmiersprache ObjectCurry in Kapitel 3 vorgestellt. ObjectCurry erlaubt die Definition von Templates, welche die Rolle von Klassen in konventionellen objektorientierten Programmiersprachen spielen. Zwischen Templates ist Einfachvererbung möglich. Objekte sind Instanzen von Templates, besitzen einen Zustand, reagieren auf Nachrichten und werden nebenläufig ausgewertet. Realisiert werden Objekte durch die in Kapitel 2 präsentierte Technik, indem ein ObjectCurry-Programm nach Curry übersetzt wird. Der spezielle Objektbezeichner `self` erlaubt einem Objekt das Versenden von Nachrichten an sich selbst. Über `super` kann auf die durch Überschreiben einer Methode verborgene Implementierung des Vorfahren (Oberklasse) zugegriffen werden. Nachrichten sind Elemente eines abstrakten Datentyps und werden ausschließlich über einen fest mit einem Objekt verbundenen Port an dieses geschickt. Subtyppolymorphismus erlaubt die Mischung von Objekten und Nachrichten verschiedener Typen entsprechend der üblichen Konformitätsregeln. Die Grundidee zur Realisierung von Subtyppolymorphismus ist die Verwendung eines einzigen Datentyps zur Aufnahme aller Nachrichten und die Verwendung eines zusätzlichen Typcheckers zur Gewährleistung der Konformität. Die formale Syntax von ObjectCurry wurde in Form von EBNF-Regeln angegeben.

In Kapitel 4 haben wir uns der verteilten Programmierung gewidmet. Nach Einführung externen Ports sind spezielle Funktionen zur Erzeugung externer Objekte eingeführt worden. Externe Objekte können Nachrichten von fremden Rechnern bzw. Prozessen empfangen. Verteilte Anwendungen erfordern häufig die Synchronisation verschiedener Prozesse. Aus diesem Grund haben wir ein Monitor-Konzept in ObjectCurry in Form der Anweisungen `delay` und `continue` eingeführt, welche den aufrufenden Prozess in eine Warteschlange einfügen bzw. einen Prozess aus einer Warteschlange aufwecken. Realisiert wurde dies durch das Verschicken einer zusätzlichen logischen Variable. `wait` r bewirkt, dass diese nicht gebunden wird, sondern in die Warteschlange r eingefügt wird. Dadurch wird der sendende Prozess suspendiert. Die Anweisung `notify` bindet die erste Variable der Warteschlange an einen speziellen Wert, der bewirkt, dass der Prozess, der auf die Bindung dieser Variable wartet, aufgeweckt wird und seine zuletzt gesendete Nachricht wiederholt.

Mit dem Typsystem von ObjectCurry haben wir uns in Kapitel 5 beschäftigt. Nach einer Darstellung der unterschiedlichen Arten von Polymorphismus, sind wir zunächst auf das Typsystem von Curry eingegangen. Typregeln erstatten die Definition von wohlgetypten Curry-Programmen. Anschließend haben wir den Begriff des eingeschränkten Typs eingeführt. Ein eingeschränkter Typ ist ein Typ verbunden mit einer Menge von Typconstraints, wobei wir nur Subtypconstraints benötigen. Durch die Vorgabe spezieller eingeschränkter Typen für Nachrichten und die Funktionen `sendMsg` und `new` kann der in Kapitel 3 gewünschte Subtyppolymorphimus erreicht werden. Die Angabe von Inferenzregeln gestattete uns eine formale Definition von Wohlgetyptheit in ObjectCurry-Programmen. Diese erfordert die Erfüllbarkeit der Constraints der berechneten eingeschränkten Typen aller im Programm definierten Bezeichner. Eine Menge von Constraints ist erfüllbar, falls eine Substitution exisitert, die alle Constraints bzgl. einer vorgegebenen Vererbungshierachie erfüllt. Die Vererbungshierachie ist durch die im Programm angegebenen Vorfahr-Subtemplate-Beziehungen bestimmt. Ein Inferenzalgorithmus sowie ein Erfüllbarkeitstest vervollständigen das Kapitel. Es ist uns gelungen, Korrektheit und Vollständigkeit des Erfüllbarkeitstests nachzuweisen.

Wesentlicher Bestandteil der vorliegenden Arbeit ist ein Compiler für ObjectCurry. Dieser übersetzt ObjectCurry-Programme nach Curry und führt eine zusätzliche Typprüfung zur Sicherstellung der Konformität zwischen Objekten und Nachrichten durch. Der Compiler besteht aus vier Modulen, nämlich Scanner, Parser, Codegenerator und Typchecker. Kapitel 6 beschreibt die Besonderheiten der Implementierung dieser Module. Der Parser wurde mit Hilfe von nicht-deterministischen Parserkombinatoren erstellt. Diese nutzen die Möglichkeiten des Nicht-Determinismus zur Darstellung der Disjunktion und freie Variablen zur Rückgabe des erzeugten abstrakten Syntaxbaums. Ein zusätzliches Ergebnis dieser Arbeit war die Erweiterung dieser Parserkombinatoren um Mechanismen zur Behandlung der Offside-Rule. Durch Bereitstellung von drei verschiedenen Sequenzoperatoren wurde dies sehr elegant erreicht. Der Codegenerator erzeugt ein Curry-Programm, das z. B. unter Verwendung des PAKCS ausgeführt werden kann. Durch die Verwendung der in den Kapiteln 2 und 3 vorgestellten Technik konnte erreicht werden, dass der erzeugte Code klar strukturiert und auch für Menschen gut lesbar ist. Der Typchecker basiert auf „Typing Haskell in Haskell "von Mark Jones. Das dort vorgestellte Programm wurde um Subtypconstraints erweitert. Eine zusätzliche Routine überprüft die Erfüllbarkeit der Subtypconstraints aller berechneten Typen.

Es wurde darauf verzichtet, parametrisch polymorphe Templates in das Typsystem zu integrieren. Eine Erweiterung um parametrischem Polymorphismus über Nicht-Objekttypen sollte jedoch mit überschaubarem Aufwand realisiert werden können: Hierzu ist es lediglich notwendig, mehrstellige Templatetypen zuzulassen.

Nicht implementiert wurden in ObjectCurry Zugriffsbeschränkungen, wie sie beispielsweise in Java über die Schüsselworte `private`, `public` etc. erfolgen. Derzeit sind alle Methoden eines Objekts nach außen sichtbar, und alle geerbten Attribute können in Subtemplates verwendet werden.

Darüber hinaus wäre es interessant zu untersuchen, ob sich die vorgestellten Konzept auch um Mehrfachvererbung erweitern lassen und welche Probleme dabei enstehen.

A. Der ObjectCurry-Compiler

Bestandteil dieser Arbeit ist ein Compiler zur Übersetzung von ObjectCurry-Programmen. Er befindet sich auf der beiligenden CD. Der Compiler wurde mit Hilfe des **PAKCS** (s. [H$^+$00]) entwickelt und ist vollständig in Curry geschrieben. Er liest den Source-Code eines ObjectCurry-Programms ein und übersetzt dieses in reines Curry. Das so erzeugte Programm kann mit einem herkömmlichen Curry-Compiler[1] weiterverarbeitet werden.

Übersetzen von ObjectCurry-Programmen

Der ObjectCurry-Compiler wurde mit Hilfe des Portland Aachen Kiel Curry System entwickelt und durch den Curry2Prolog-Compiler nach Prolog übersetzt. Ein gespeicherter Zustand *(saved state)* befindet sich unter dem Namen `occ` auf der CD. Auf einem System mit einem installierten Sixtus Prolog Compiler sind zur Übersetzung eines ObjectCurry-Programms daher folgende Schritte notwendig:

1. Starten des ObjectCurry-Compilers mit `occ`
2. Übersetzen des ObjectCurry-Programms mit `c "`*filename*`"`, wobei *filename*`.ocurry` der Name des zu übersetzenden Programms ist.

Anschließend liegt das übersetzte Programm als Datei mit dem Namen *filename*`.curry` vor und kann mit Hilfe des PACKS übersetzt und ausgeführt werden:

- Starten des Curry2Prolog-Compilers mit `curry2prolog`
- Compilieren des Programms mit `:l` *filename*

Quellcode neu compilieren

Um den Quellcode des ObjectCurry-Compilers neu zu compilieren, werden die folgenden Dateien benötigt:

- `scanner.curry`
- `parser.curry`
- `builder.curry`

[1] beispielsweise dem Curry2Prolog-Compiler des PAKCS

- `typing.curry`
- `ocurrytyping.curry`
- `transitiv.curry`
- `additionals.curry`
- `main.curry`

Es genügt, das Modul `main.curry` zu übersetzen, da dieses alle übrigen Module importiert.

B. Beispiele

Auf der beiligenden CD finden sich neben dem ObjectCurry-Compiler auch eine Reihe von Beispielprogrammen, die die wichtigsten Konzepte von ObjectCurry demonstrieren:

`counter.ocurry`	Der Zähler aus Kapitel 3. Ein Aufruf der Funktion `test` erzeugt ein `MaxCounter`-Objekt, sendet einige Nachrichten an dieses und liefert den Wert 2 zurück.
`counter2.ocurry`	Einige Funktionen, deren Typen nicht nicht-erfüllbare Constraints enthalten. Die Funktionen `notAllowed`, `notAllowed2` und `notAllowed3` besitzten Typen, deren Constraints nicht erfüllbar sind.
`counter3.ocurry`	Ein externer Zähler. `server` startet ein `Counter`-Objekt und veröffentlicht dieses. Mit `client n` kann auf das externe `Counter`-Objekt zugegriffen werden, wobei `n` der Name ist, unter dem dieses veröffentlicht wurde.
`buffer.ocurry`	Der Puffer aus Kapitel 4. `test` liefert den Wert `[0,1,2]`.
`hoo.ocurry`	Ein Objekt höherer Ordnung: Bei Erhalt der Nachricht `SendInc o` schickt dieses dem Objekt `o` die Nachricht `Inc`.
`super.ocurry`	Beispiel zur Verwendung von `super`.
`dynamic.ocurry`	Das Beispiel aus Kapitel 3.2.3.

Index

Abbildungsverzeichnis

Literaturverzeichnis

[ASU90] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullmann. *Compilerbau, Teil 1*. Addison-Wesley, Bonn, 1990.

[Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, April 1987.

[CLF99] Rafael Caballero and Francisco J. Lopez-Fraguas. A functional logic perspective of parsing. In *Proc. FLOPS'99*, pages 85–99. Springer LNCS 1722, 1999.

[cor] *The CommonObjectRequestBroker: Architecture and Spezifikation*, 1.1 edition. OMG Document Number 91.12.1.

[CW93] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. In *ACM Computing Surveys*, volume 17, 1993.

[CW97] Mary Campione and Kathy Walrath. *Das Java Tutorial: Objektorientierte Programmierung für das Internet*. Addison-Wesley, Bonn, 1997.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[H+00] Michael Hanus et al. The Portland Aachen Kiel Curry System curry, 2000. `http://www.informatik.uni-kiel.de/~pakcs`

[Han87] Michael Hanus. *Problemlösen mit Prolog*. B. G. Teubner, Stuttgart, 1987.

[Han97] Michael Hanus. A unified computation model for functional and logic programming. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 80–93, 1997.

[Han98] Michael Hanus. Prinzipien von Programmiersprachen, August 1998. Vorlesung, gehalten im WS 98/99 an der RWTH-Aachen.

[Han99] Michael Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

[Han00a] Michael Hanus. Curry: An Integrated Functional Logic Language, 2000. `http://www.informatik.uni-kiel.de/~curry/report.html`

[Han00b] Michael Hanus. A functional logic programming approach to graphical user interfaces. In *Proc. of the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.

[HS95] John Hughes and Jan Sparud. Haskell++: An object-oriented extension of Haskell. 1995.

[Inp96] Inprise, editor. *Delphi 5: ObjectPascal Programmierhandbuch.* Borland GmbH, Langen, 1996.

[JCJÖ97] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering.* Addison-Wesley, Essex, England, 1997.

[JMH93] S. Janson, J. Montelius, and S. Haridi. Ports for objects in concurrent logic programs. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 211–231. MIT Press, London, 1993.

[Jon99] Mark P. Jones. Typing Haskell in Haskell, 1999.
`http://www.cse.ogi.edu/~mpj/thih/`

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *1992 ACM Conference on Lisp and Functional Programming*, pages 193–204. ACM, ACM, August 1992.

[KTMB87] K. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow. Vulcan: Logical concurrent objects. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers (Volume II)*, pages 274–303. MIT Press, London, 1987.

[Ler96] Xavier Leroy. The objective caml system. Technical report, 1996.

[LFSH99] F. J. Lopez-Fraguas and J. Sanchez-Hernandez. Toy: A multiparadigm declarative system. In *Proc. RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

[Nag88] Manfred Nagl. *Einführung in die Programmiersprache Ada.* Vieweg, Braunschweig, 1988.

[Nor99a] Johand Nordlander. Rationale for O'Haskell, August 1999.
`http://www.cs.chalmers.se/~nordland/ohaskell/rationale.html`

[Nor99b] Johand Nordlander. *Reactive Objects and Functional Programming.* PhD thesis, Chalmers Göteborg University, May 1999.

[OW97] M. Odersky and P. Wadler. Pizza into java: Translating theory into practice. In *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL)*, pages 146–159, 1997.

[P+97] J. Peterson et al. Haskell: A non-strict, purely functional language (version 1.4). Technical report, Yale University, Yale, 1997.

[rmi00] Java Remote Method Invocation, April 2000.
`http://java.sun.com/products/jdk/rmi/`

[SG94] Abraham Silberschatz and Peter B. Galvin. *Operating Systems Concepts.* Addison-Wesley, Reading, Massachusetts, 1994.

[ST83] E. Shapiro and A. Takeuchi. Object oriented programming in concurent prolog. *New Generation Computing*, 1:25–48, 1983.

[Thi94] Peter Thiemann. *Grundlagen der funktionalen Programmierung.* B. G. Teubner, Stuttgart, 1994.

[vG93] J. Wolff von Gudenberg. *Objektorientiert programmieren von Anfang an.* BI-Wiss.-Verl., Mannheim, 1993.

[W+95] Dieter Werner et al. *Taschenbuch der Informatik.* Fachbuchverlag Leipzig, Leipzig, 1995.