INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Sprachen, Architekturen und neue objektorientierte Softwaretechniken

Bad Honnef, 7.-9. Mai 2001

Wolfgang Goerigk, Elke Pulvermüller
und Andreas Speck (Hrsg.)

Bericht Nr. 2018
Oktober 2001

PAX OPTIMA RERUM

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstraße 40, D-24098 Kiel
Germany

Sprachen, Architekturen und neue objektorientierte Softwaretechniken

Physikzentrum Bad Honnef

7. – 9. Mai 2001

**Wolfgang Goerigk, Elke Pulvermüller
und Andreas Speck (Hrsg.)**

wg@informatik.uni-kiel.de

**Bericht Nr. 2018
Oktober 2001**

Dieser Bericht enthält eine Zusammenstellung der Beiträge des gemeinsamen Workshops der GI-Fachgruppen 2.1.4 und 2.1.9. Der Bericht ist als persönliche Mitteilung aufzufassen.

Vorwort

Die GI-Fachgruppe 2.1.4 *Programmiersprachen und Rechenkonzepte* veranstaltete gemeinsam mit der Fachgruppe 2.1.9 *Objektorientierte Softwareentwicklung (OOSE)* vom 7. bis 9. Mai 2001 im Physikzentrum Bad Honnef ihren 18. Workshop, in diesem Jahr zum Thema *Sprachen, Architekturen und neue objektorientierte Softwaretechniken*. Der Workshop förderte gegenseitiges Kennenlernen, Erfahrungsaustausch, intensive Diskussion und die Vertiefung gegenseitiger Kontakte und gemeinsamer Interessen beider Fachgruppen.

Der vorliegende Tagungsband stellt die Beiträge zu diesem Workshop zusammen, die über neue Arbeiten und Ergebnisse aus den Interessensbereichen beider Fachgruppen berichten und deren Qualität und Vielfalt zu einem sehr interessanten Programm beigetragen haben. Schwerpunkte lagen auf den Gebieten der objektorientierten und deklarativen Softwaretechniken und der Programmierung und Beherrschung nebenläufiger und verteilter Systeme, und die Themen reichten von theoretischen und programmiersprachlichen Grundlagen, Semantik, Verifikation über architekturzentrierte und komponentenbasierte Systementwicklung bis hin zur Distribution mobilen Codes und sehr praktischen Anwendungen im World Wide Web. Unter anderem waren objektorientierte, funktionale, funktional-logische Sprachen und Datenstrukturen, Architekturen, generative Softwareentwicklung, Komponentensysteme und Features, Verifikation und formale Beschreibung nebenläufiger Systeme und auch empirische Untersuchungen zu Programmoptimierung Themen, die Anlass zu Erfahrungsaustausch und Fachgesprächen gaben.

Allen Teilnehmern möchten wir dafür danken, dass sie mit ihren Vorträgen und konstruktiven Diskussionsbeiträgen zum Gelingen des Workshops beigetragen haben. Dank für die Vielfalt und Qualität der Beiträge gebührt den Autoren. Ein Wort des Dankes ebenso an die Mitarbeiter und die Leitung des Physikzentrums Bad Honnef für die gewohnte angenehme und anregende Atmosphäre und umfassende Betreuung. Der Technischen Fakultät der Christian-Albrechts-Universität zu Kiel möchten wir dafür danken, dass sie den Druck des Tagungsbandes ermöglicht hat.

Kiel, Karlsruhe, Jena im Oktober 2001

Wolfgang Goerigk
Elke Pulvermüller
Andreas Speck

Inhaltsverzeichnis

Generative Software-Entwicklung mit UML

Christian Donker
Thomas Stahl

b+m Informatik AG, D-24109 Melsdorf, Germany

Wolfgang Goerigk

Institut für Informatik und Praktische Mathematik
Technische Fakultät der Christian-Albrechts-Universität zu Kiel

0 Einleitung

Dieses Dokument liefert einen kurzen Einblick in die generative Software-Entwicklung mit UML. Der Ansatz der architekturzentrierten Software-Entwicklung ist eine wichtige Voraussetzung für die generative Software-Entwicklung, um die generierbaren Anteile identifizieren und spezifizieren zu können.

Wir werden die Vorteile der architekturzentrierten Software-Entwicklung erörtern sowie Grundlagen der generativen Software-Entwicklung erklären.

1 Architekturzentrierte Software-Entwicklung

Architekturzentrierte Software-Entwicklung ist als Generalisierung der generativen Software-Entwicklung zu betrachten. Bei der architekturzentrierten Software-Entwicklung wird ein Vorgehensmodell definiert; die generative Software-Entwicklung ist dann eine Ausprägung dieser unter Verwendung eines Generator-FrameWorks.

Im Mittelpunkt der architekturzentrierten Software-Entwicklung stehen sogenannte frei konfektionierbare, musterbasierte *Anwendungsfamilien*. Sie beinhalten fundiertes Spezialwissen, welches einer breiten Schicht von Anwendungsentwicklern zur Verfügung gestellt wird und als Grundlage verschiedener ähnlicher Entwicklungen dienen sowie weiterentwickelt und verbessert werden kann. Als *ähnlich* bezeichnen wir Anwendungen einer Anwendungsfamilie, die also im wesentlichen von gleicher Architektur sind.

1.1 Anwendungsarchitektur

Mit Anwendungsarchitekturen sind an dieser Stelle die prinzipiellen *horizontalen* Strukturen (Muster) und Schichten, wie z. B. ein Komponentenbegriff Model-View-Controller Schichtung einer Anwendung gemeint und nicht *vertikale* Strukturen, wie z. B. der konkrete Komponentenschnitt einer Anwendung. Eine Anwendungsarchitektur kann somit eine ganze *Familie* von Anwendungen mit gleichen Konzepten tragen; wiederkehrende Problemstellungen werden auf eine einheitliche Art und Weise behandelt.

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

1.2 Architekturzentriertes Design

Die Verwendung von architekturspezifischen Design-Mustern (siehe auch [3]) wird als *architekturzentriertes Design* bezeichnet. Die Notation erfolgt mittels der UML™ unter der Verwendung von Stereotypen, siehe auch [4]. Auf diese Weise entsteht eine anwendungsarchitekturspezifische, semantisch stark angereicherte UML-Design-Sprache. Diese Design-Sprache hat die Aufgabe, den Weg von der Analyse zum Design, sowie vom Design zur Implementierung zu erleichtern und zu beschleunigen.

Bei einem klassischen objektorientierten Entwicklungsprozeß entsteht aus dem Analyse-Modell iterativ und inkrementell ein Design-Modell, welches so weit verfeinert und detailliert ist, daß es durch einen Standard Code-Generator in einen Implementierungsrahmen überführt werden kann. Ein solches Design-Modell wird auch im folgenden mit *Implementierungsmodell* bezeichnet.

Abbildung 1 stellt eine Scheibe des Wasserfallmodells dar (Mikrowasserfall). Die Y-Achse trägt als Einheit den Detaillierungsgrad, die X-Achse die des Informationsgehaltes. Auf dem Weg zur Implementierung ist ein kontinuierlicher Informationsgewinn, sowie eine Erhöhung des Detaillierungsgrades zu erwarten. Die Schraffur ist als Aufwand zu interpretieren.

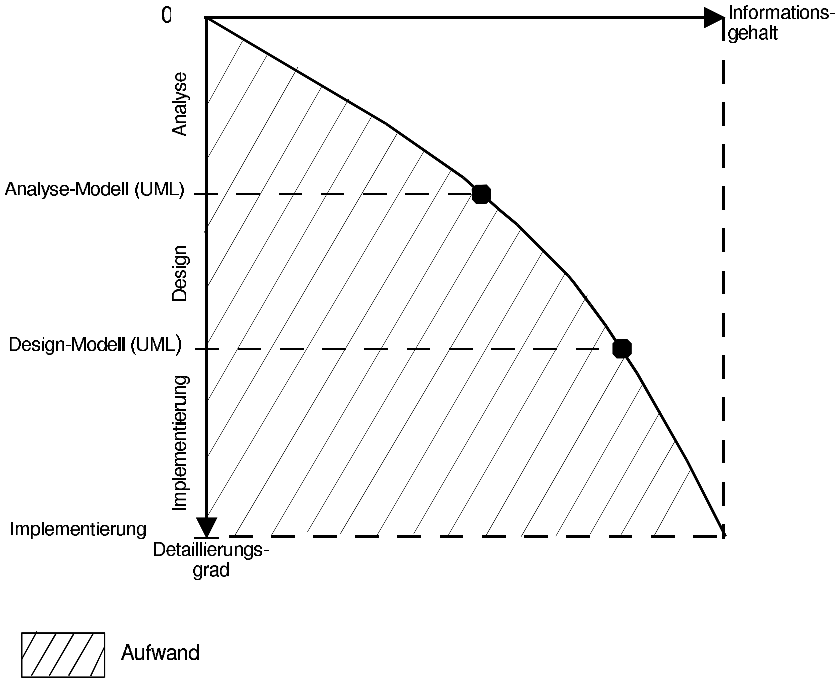


Abbildung 1. Informationsgewinn / Standard OO-Prozeß (aus [1])

Der Weg vom Analyse-Modell zum Design-Modell ist im allgemeinen sehr lang, und es gibt keine wohldefinierten Zwischenstopps. Des weiteren existieren Differenzen und Uneinheitlichkeiten in den Ergebnissen unterschiedlicher Designer.

Das Problem dabei ist, daß es zu keinem nutzbaren Gesamtmodell führt. Diesem Umstand sind nur intensives Coaching und Design-Reviews entgegenzusetzen.

Offensichtlich ist auch, daß ein so konzipiertes, konventionelles Design-Modell enorm detailliert ist. Die Kerninformationen gehen in dieser Fülle von Details unter.

Eine weitere unangenehme Eigenschaft eines solchen Implementierungsmodells ist, daß es sehr anfällig im Bezug auf Design-Änderungen ist. Das hat zur Folge, daß genau diese Änderungen vorzugsweise im Quelltext der Anwendung durchgeführt werden und dann das aus diesem Code resultierende, reverseengineerte Design-Modell als konsistente Dokumentation genommen wird, bzw. als Modell des nächsten Inkrements. Dieses Vorgehen wird deshalb auch als *Roundtrip-Engineering* bezeichnet, siehe dazu auch [7].

Packt man die Architekturmerkmale einer Anwendung in eine Design-Sprache und nutzt eine solche, um das Design-Modell zu entwerfen, so entsteht ein Design-Modell geringeren Detaillierungsgrades mit einem höheren Informationsgehalt, was deutlich in Abbildung 2 zu erkennen ist.

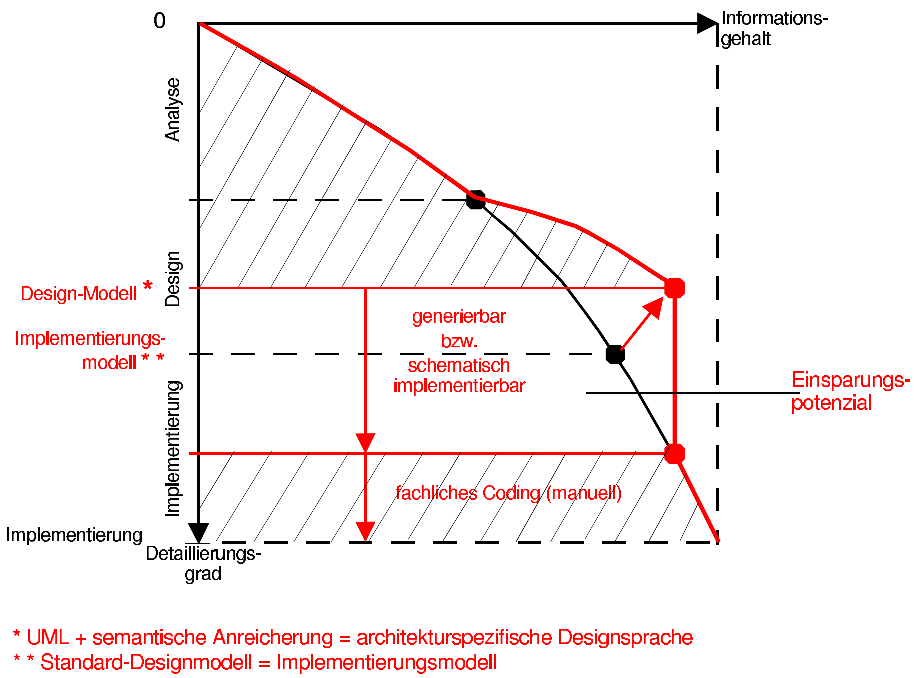


Abbildung 2. Informationsgewinn / Architekturzentrierter OO-Prozeß (aus [1])

Die Aufwärtsbewegung des Punktes des Design-Modells resultiert aus der Tatsache, daß mit der Design-Sprache komplexe Architekturmuster bzw. Design-Muster mit knapper UML-Notation ausdrückbar sind; z. B. impliziert die Verwendung eines Stereotyps `<<Entity>>` einer Enterprise JavaBean^{TM} Design-Sprache eine Fülle von Klassen und Beziehungen, welche durch eine einzige knappe Benennung ausdrückbar ist.

Die Tatsache, daß die Semantik einer Design-Sprache auch eine Implementierung beinhaltet, hat die Rechtsbewegung des Punktes des Design-Modells zur Folge.

Ist nun der Ausgangspunkt ein Design-Modell, basierend auf einer Design-Sprache plus einer festen Architektur, so läßt sich die Implementierung des Design-Modells in zwei Teile zerlegen. Zuerst kann die Implementierung erfolgen, konzentriert auf die architektonischen Bestandteile der Anwendung. Dies erfolgt mittels der Bedeutung der Design-Sprache; die Umsetzung entspricht einem reinen „Funktionieren" der Implementierer. Der zweite Teil der Implementierung beschäftigt sich mit der Realisierung der sogenannten Fachlogik; dieser Abschnitt wird mit *fachlichem Coding* bezeichnet.

Betrachtet man in Abbildung 2 den Übergang vom Design-Modell zur Implementierung, so ist zu erkennen, daß die Implementierung der architektonischen Bestandteile der Anwendung, ausgehen vom Design-Modell kein Informationsgewinn bedeutet, da genau diese Informationen schon in der Semantik der Design-Sprache enthalten sind.

Ein weiterer Punkt, welcher nicht unerwähnt bleiben sollte, ist die optimale Trennung von Architektur, Design und fachlicher Implementierung. Dies impliziert ein klares und sauberes Rollenmodell mit entsprechenden Verantwortlichkeiten im Entwicklungsprozeß.

1.3 Syntax und Semantik der Design-Sprache

Die Syntax der Design-Sprache wird beschrieben durch eine Menge von erlaubten Stereotypen sowie eine Menge von Zusicherungen. Die Zusicherungen beschreiben, welche Stereotypen an welche UML-Modellelemente notiert werden dürfen und welcher Kontext vorhanden sein muß; z. B. daß ein Stereotyp `<<Key>>` nur an Attributen in einer Klasse mit dem Stereotyp `<<Entity>>` verwendet werden darf.

Die Semantik einer Design-Sprache wird unter Zunahme einer Architektur mittels eines Paares Referenz-Design und Referenz-Implementierung ausgedrückt. Das Referenz-Design umfaßt, soweit möglich, alle Elemente der Design-Sprache in einem sinnvollen Kontext, sowie alle möglichen Konstellationen. Die Referenz-Implementierung ist „die" Implementierung des Referenz-Designs zu der gegebenen Architektur. Durch diese Implementierung wird ein Programmiermodell vorgegeben, sowie die Interpretation der einzelnen Design-Elemente durch die entsprechende Bedeutung der Implementierung bzw. der Architektur.

Um Eindeutigkeit bei der Abbildung des Referenz-Modells auf die Referenz-Implementierung zu erhalten, sind sogenannte Projektionshilfen erforderlich, welche im allgemeinen als textuelle Informationen abgelegt werden; es handelt sich dabei um Quelltextfragmente aus der Referenz-Implementierung und eine Beschreibung, aus welchen Teilen des Referenz-Designs diese resultieren.

1.4 Anwendungsfamilie

Als *Anwendungsfamilie* bezeichnen wir ein Tupel, das aus einer architekturspezifischen Design-Sprache und einer konkreten Anwendungsarchitektur besteht. Mit Hilfe dieses Begriffs ist es möglich, auf eine elegante Art die architekturellen Gemeinsamkeiten ähnlicher Anwendungen zu extrahieren.

2 Generative Software-Entwicklung

Ausgehend von einem Design-Modell, spezifiziert mittels einer UML-Design-Sprache, kann aus diesem Modell die vollständige Implementierung der architektonischen Bestandteile erzeugt werden, wenn genau diese Informationen über die zu verwendende Architektur in Form von einer Anwendungsfamilie vorliegen.

In diesem Generat sind sogenannte *Protected Regions* vorgesehen, welche dann von einem Implementierer mit Fachlogik gefüllt werden. Beim Generieren ist es des weiteren möglich, ein Generat eines Vorgängerinkrements mit in den Generierungsprozeß einzubringen, um inkrementelle Entwicklung zu gewährleisten. Die Fachlogik aus einem Generat bleibt somit beim nächsten Generieren erhalten.

Dadurch, daß das Generat nur an gewissen, ausgezeichneten Markierungen das Hinzufügen von Fachlogik zuläßt, ist der Implementierer auf ein gegebenes Programmiermodell festgelegt. Des weiteren sind Design-Änderungen nicht mehr im Generat bzw. Quelltext durchführbar, diese <u>müssen</u> im Design vollzogen werden.

Außerdem ist durch diesen Ansatz eine sehr gute Wartbarkeit erreicht, da z. B. eine horizontale Änderung der Architektur nicht mehr in der vollen Breite der Anwendung nachvollzogen werden muß, sondern nur noch an genau einer Stelle, in der Anwendungsfamilie, und der Generierungsprozeß aus (vgl. Abschnitt 2.5) nur neu angestoßen werden muß.

2.1 Generator-FrameWork

Das Generator-FrameWork gehört zu der Klasse der sogenannten *White box frameworks*, siehe auch [9], d. h. eine gewisse Anzahl von Klassen müssen spezialisiert werden, um so die anwendungsfamilienspezifischen Anteile einzubringen. Dieser dynamische Anteil wird über den objektorientierten Polymorphismus zur Laufzeit an den generischen Teil des Generator-FrameWork gebunden; dies ist möglich, da als Entwicklungsumgebung Java™ gewählt wurde, was selbstverständlich <u>keinerlei</u> Auswirkungen auf die Zielsprache der Anwendung hat.

Um ein Design-Modell in Quelltext zu überführen, sind gewisse Informationen nötig, zum einen über die Design-Sprache und zum anderen, wie diese Elemente der Design-Sprache expandiert werden sollen. Diese Informationen bilden die Anwendungsfamilie. Die Definition der Design-Sprache wird mittels Spezialisierungen von FrameWork-Klassen realisiert. Die Quelltextfragmente sowie Expansionsregeln sind in sogenannte *Templates* verpackt (siehe 2.2).

2.2 Metamodell und Anwendungsfamilie

Eine Anwendungsfamilie ist definiert über eine Design-Sprache und eine konkreten Architektur. Technisch besteht die Anwendungsfamilie im Generator-FrameWork aus einer Menge von Spezialisierungen von Generatorklassen, welche aus der Definition der Design-Sprache resultieren und aus Templates, welche eine Implementierung der Architekturbestandteile enthalten. Somit sind die anwendungsfamilienspezifischen Anteile im Generator-FrameWork allgemein für alle Mitglieder dieser Familie von Anwendungen.

Spezielle Generatorklassen Die Spezialisierungen der Generatorklassen definieren die Design-Sprache. Über diese Klassen werden die erlaubten Konstrukte, Stereotypen spezifiziert, sowie die notwendigen Zusicherungen, welche von jedem Satz der Sprache erfüllt sein müssen.

Zusätzlich bieten diese Spezialisierungen dem Anwendungsfamilienentwickler die Möglichkeit, Funktionalitäten dem Template-Programmierer zur Verfügung zu stellen, falls die Expansion der Templates weitere Informationen aus dem Design-Modell benötigen.

Templates Die Templates einer Anwendungsfamilie bestehen aus einer Menge von Definitionen, gebunden an ein Modell-Element der UML plus Identifikator (Name des Templates). Diese Template-Definitionen sind angereichert mit Sprachelementen, welche für die Navigation auf Graphen und die Expansion von Zeichenreihen eigens entworfen worden sind.

Der Template-Programmierer ist somit in der Lage, ein Design-Modell zu analysieren und in Abhängigkeit dieses Modells Quelltext zu erzeugen. Zusätzlich hat der Programmierer innerhalb der Templates die Möglichkeit, Funktionalitäten aus den anwendungsfamilienspezifischen Generatorklassen zu nutzen. Somit ist eine mächtige und praxistaugliche Umgebung geschaffen, um genau die Implementierung der angewandten Muster im Design-Modell zu erzeugen.

2.3 Anwendungsdesign

Ein Anwendungsdesign als Satz einer Design-Sprache muß genau die Zusicherungen der Design-Sprache erfüllen, und es dürfen nur die definierten Elemente (Stereotypen) verwendet werden. Ein solches Anwendungsdesign enthält mehr Informationen bei geringerem Detaillierungsgrad, als ein konventionelles UML-Design-Modell, siehe dazu auch 1.2. Die Bedeutung eines Anwendungsdesigns, gebildet mittels einer architekturspezifischen Design-Sprache, ist somit über die Anwendungsfamilie (siehe 2.2) definiert.

Um dieses Design-Modell dem Generator-FrameWork zugänglich zu machen, wird es in XML/XMI (siehe [5] und [8]) überführt. Im Hause b+m wird zur Zeit mit dem UML-Werkzeug Rational Rose™ gearbeitet, welches eine XMI-Exportfunktionalität zur Verfügung stellt.

2.4 Der generische Generator

Der generische Generator des Generator-FrameWorks ist allgemein für alle Anwendungen sowie für alle Anwendungsfamilien. Dieser umfaßt die Logik, die nötig ist, um Elemente der Design-Sprache mittels der Templates der Anwendungsfamilie zu expandieren und die Ausgabe zu produzieren. Die anwendungsfamilienspezifischen Anteile werden zur Laufzeit an den Generator gebunden; dies sind genau jene Spezialisierungen der Generatorklassen aus der Anwendungsfamilie (siehe 2.2).

Die Funktionalität des generischen Generators umfaßt weiterhin die Möglichkeit, Quelltext eines Vorgängerinkrementes mit einzubeziehen und diesen Quelltext genau in die „richtige " Position einzumischen, so daß sichergestellt ist, daß bei Neugenerierung keine Implementierungen verloren gehen.

2.5 Die Quelltextgenerierung

Das Anwendungsdesign wird in XMI überführt (siehe dazu [5]). XMI dient der standardisierten Darstellung eines UML-Satzes in XML (siehe dazu [8]). Diese XML/XMI-Repräsentation des Design-Modells erhält der generische Generator zusammen mit den Templates als Eingabe. Zusätzlich wird der Generator um spezialisierte Generatorklassen erweitert, in denen der anwendungsfamilienspezifische Anteil der Design-Sprache kodiert ist. Des weiteren enthalten diese Spezialisierungen Methoden, welche aus den Templates angesprochen werden können. Optional kann bei Generierung die Anwendung mit Quelltexten eines Vorgängerinkrements angereichert werden.

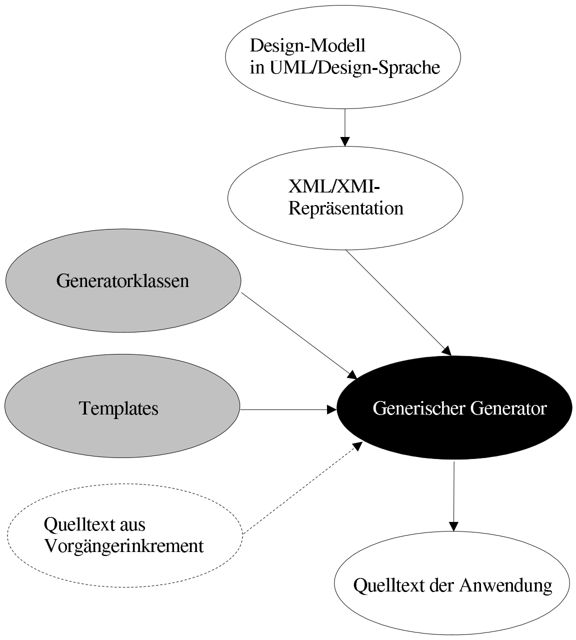


Abbildung 3. Ablauf

Der Ablauf der Generierung ist in Abbildung 3 dargestellt. Der generische Anteil ist schwarz dargestellt, der der Anwendungsfamilie grau.

3 Zusammenfassung und Erfahrungen

Der Ansatz der architekturzentrierten Software-Entwicklung ist im Hause b+m verfeinert und ausgearbeitet worden und wird in diversen Projekten mit großem Erfolg praktiziert. Durch den Einsatz einer Design-Sprache wird Expertenwissen für eine breite Schicht von Anwendungsentwicklern formal verfügbar gemacht, die dieses Wissen im Detail nicht mehr benötigen.

Durch den Einsatz eines Generators ist das Programmiermodell festgelegt, da in der Phase der Implementierung der Programmierer keine Design-Änderungen vornehmen kann, sondern diese im Design durchgeführt werden müssen. Es handelt sich somit um reines *Forward-Engineering*. Die nötige Flexibilität wird durch Einfachheit der Modifikation und Verbesserung des Designs und durch Schichtung und Modularisierung des gesamten Entwicklungsprozesses erreicht.

Das Generator-FrameWork wird zur Zeit bei zwei Projekten der Größe von 1000-2000 Personentagen eingesetzt. Es hat sich gezeigt, daß sich der Entwurf einer eigenen Designsprache und die Entwicklung spezieller Templates bezogen auf eine Anwendung dieser Größenordnung rentiert.

Die Erfahrungen zeigen, daß die architekturspezifischen Anteile, welche generierbar sind, bis zu 70% des gesamten Codes der Anwendung ausmachen können.

Literatur

1. b+m Informatik AG, `http://www.bmiag.de/de/_frame/b+m/kompetenz.htm`. *Software-Entwicklung bei b+m*, Juni 2001.
2. C. Donker. Generative Softwareentwicklung mit UML: Vom Design zum Codegenerator. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, 2000.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster*. Addison Wesley Longman Verlag GmbH, 1996.
4. OMG, `http://www.omg.org/technology/documents/formal/unified_modeling_language`. *Unified Modeling Language (UML)*, 1.3 edition, März 2000.
5. OMG, `http://www.omg.org/technology/documents/formal/xml_metadata_interchange`. *XML Metadata Interchange (XMI)*, 1.1 edition, November 2000.
6. I. Pavković. Generative Softwareentwicklung mit UML: Codegenerierung. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, 2000.
7. B. Steppan. Prozessoptimierung, Roundtrip-Engineering mit Together 4.1 und JBuilder 4. *ix, Magazin für professionelle Informationstechnik*, 8:118, August 2001.
8. W3C, `http://www.w3.org/TR/2000/REC-xml-20001006`. *Extensible Markup Language (XML)*, 1.0 edition, Oktober 2000.
9. S. Zamir. *Handbook of Object Technology*. CRC Press LLC, 1999.

Lava – An Object-Oriented RAD Language Designed for Ease of Learning, Use, and Program Comprehension

Klaus D. Günther[1], Irmtraut Günther[1]

[1] GMD, Institute for Secure Telecooperation, Rheinstr. 75,
D-64295 Darmstadt, Germany
{Klaus.Guenther, Irmtraut.Guenther}@darmstadt.gmd.de

Abstract. The growing demand for new software calls for a considerable acceleration of the software production process and for a sensible relaxation at the software maintenance front. These goals can be achieved only if we can decisively increase the degree of modularity, variability, comprehensibility of software, or short: the degree of structured programming, as well as the simplicity of program manipulation, restructuring, and transformation. The experimental object-oriented language "Lava" and the associated programming environment "LavaPE" attempt to achieve these goals by providing quite a number of unusual features. The most remarkable features are: 1. Text editors are completely replaced with Lava-specific structure editors. 2. A Lava class consists of a public "interface" and a completely separate, exchangeable "implementation" which may be stored in a different file. 3. Frameworks and design patterns are supported in a very natural way by allowing packages and interfaces to have overridable type parameters.

1 Introduction

The continuously growing demand for new software can be supplied only if the productivity of the programming process can be decisively increased. (Cf. section 3 of the PITAC Report [10] to the American government, which assigns maximum priority to this goal.)

Moreover, the amount of work flowing into the continuous maintenance of large commercial software products throughout their life cycle can be decisively reduced if programming languages and programming environments encourage or even enforce (to some degree) a clear and natural subdivision of programs into small, self-contained, self-evident, independently comprehensible units.

All this means that we cannot be satisfied with the popular programming languages like C++ [11], Java [3], or Visual Basic. Rather, we have to aim for greatly improved languages and RAD programming environments

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

- that relieve the programmers from clerical and error-prone work,
- that in particular replace text-editors with several kinds of structure-editors,
- that are easy to learn and get along with a minimum of orthogonal concepts,
- that facilitate program comprehension,
- that facilitate clear syntactic separation of abstraction levels
  1. in the small, by favoring small, self-contained (possibly recursive) functions rather than complicated, deeply nested loop constructs,
  2. in the large, (a) by supporting nesting of declarations according to their primary or auxiliary nature, (b) by strictly separating interfaces from implementations, (c) by utilizing multiple inheritance to compose large classes in an easily configurable way from specialized small base classes that can be implemented independently,
- that clarify the control flow as well as the data flow of programs by appropriate constructs and restrictions,
- that unify what should not be separated ("embedded SQL"),
- that separate what should not be intermingled (interfaces and implementations),
- and that support reuse and multiple versions of certified components and proven design patterns from clearly documented component and pattern libraries.

The experimental programming language Lava and the associated LavaPE programming environment offer solutions to quite a number of these problems. The open-source project Lava is intended primarily to provide a public playground for experimenting with new ways of combining advanced object-oriented language features with ease of use and comprehension. Everybody is invited to participate in this explorative process. Lava is particularly attractive for those researchers who are interested in program analysis, program synthesis, and program transformation, since Lava programs are processed internally as “abstract syntax trees” all the time, beginning from their construction in specific structure editors until their execution by the Lava interpreter.

An early preview version of the Lava programming environment LavaPE (for Windows 9X/NT/2000 platforms, including a few code samples) can be downloaded from the Lava web site [8]. There you can find a more comprehensive online documentation and further papers. Cf. also [4].

Section 2 below deals with Lava contributions to the goals "ease of learning" and "ease of use". Section 3 explains how program comprehension is facilitated by quite a number of different Lava features. Section 4 outlines how Lava copes with "genericity" in a new way, based on “virtual types". These are particularly suited to provide very natural specifications of "design patterns" and "frameworks", viewed as (groups of mutually recursive) “virtualized” types. This capability will play a much more important role in future languages.

2 LavaPE and Ease of Learning / Ease of Use

2.1 Replacing Text Editors with Structure Editors

Lava programs are no longer "written" but "constructed" from basic constructs, using point/click/drag/drop/cut/copy/paste and menu selection operations, and this is true also for the executable parts of Lava programs.

The Lava programming environment LavaPE is completely based on structure editing, with two dominating, primary views: the "declaration view" and the "exec view" (see Fig.1, last page):

- The declaration view is used for declaring various kinds of Lava entities, in particular new packages, interfaces, implementations, and their respective substructures.
- The exec view is used to construct the executable portions of Lava programs, i.e.,
  1. "exec's", = bodies of functions and of "initiators" (= main programs),
  2. "constraints", which may be associated with interfaces and must be fulfilled whenever a new object supporting the respective interface has been created.

The *declaration view* is a "tree view" to which everybody is accustomed, for instance from the "Explorer" of Microsoft Windows. Declarations may be nested to any depth in Lava. Tree construction is controlled by tool buttons corresponding to the basic Lava notions, like "new package", "new interface", "new implementation", "new member variable", "new member function", "new function parameter", etc. The properties of these entities are edited using appropriate property sheets. Subtrees can be easily copied and moved by drag-and-drop operations or expanded/collapsed by specific tool buttons.

There are several auxiliary tree views the most important of which is used for specifying the details of an interface or package/pattern derivation leading to a derived interface or package/pattern. (See Fig.1, last page.)

The *exec view* is a quite normal textual representation of an "exec" or "constraint". But although it uses the standard Windows "rich edit view", it is not editable directly as text. The executable program text is rather constructed from a number of basic statement, expression, and special constructs, which would typically contain "placeholders" (= syntactic variables) <stm>, <expr>, <var>, <type>, <func>, <set> ... for statements, expressions, references... that may be inserted in these places.

In fact, there is no fixed textual syntax of Lava at all, nor is there a Lava parser or compiler. The point-and click operations of the programmer generate and manipulate an internal tree representation of the Lava program (an “AST”, short for “abstract syntax tree”) *directly*. The readable representation of declarations, execs, and constraints is generated only on the fly as long as a corresponding declaration or exec view is open.

No text entry whatsoever is required in LavaPE, except for comments, constants, and new identifiers. Syntax errors cannot occur any longer. Context-related errors are reported at the earliest possible time. References to be inserted are selected from specific combo-boxes whose current content depends largely on the current selection.

So Lava is not a conventional textual language, but it is inseparably connected with LavaPE.

2.2 Automatic Maintenance of References

Readable, textual identifiers are, in a sense, meaningless in Lava. Every Lava entity has a unique internal identifier that is never changed. All references to Lava entities are based solely on these internal identifiers. A readable textual name is associated with such an immutable, unique, internal identifier at the place where the respective Lava entity is declared, and it can be changed only there.

Since readable representations of Lava programs are generated only on the fly when they are opened in one of the Lava structure editors, all references to Lava entities will always be up to date: The textual name of a Lava entity is always "fetched" from its actual declaration and inserted at the place of reference. Textual names need not be unique even. But Lava tries to prevent you from using duplicate names in Lava, of course, since they impair the comprehensibility of programs and force you to use the "go to declaration" function of LavaPE to find the actual meaning of the respective name.

Automatic maintenance of references is a quite important advantage for source code maintenance. It happens very often that you would like to assign a more meaningful name to an existing entity, but it is extremely laborious and boring to identify all affected source files and to use string search and replacement in order to change all affected references. This will often prevent you from introducing more significant names. In Lava you need only change the name in the declaration of the respective Lava entity.

Automatic maintenance of references applies also in cases where declarations are moved (using drag-and-drop) within the Lava declaration tree or even between different files: The path-names of Lava interfaces, packages, functions, etc., that reflect the position of these entities in the declaration tree, are changed accordingly in all references to the affected entities. Cf. [1] for an alternative approach to identifier change and maintenance.

2.3 Automatic Maintenance of Function Calls

Another kind of automatic maintenance of references applies to member functions of interfaces and implementations. If you add or delete or permute formal parameters of a function then all existing invocations of these functions are changed immediately or as soon as the containing Lava file is opened: Placeholders for actual parameters are inserted where new formal parameters have been inserted; actual parameters corresponding to deleted formal parameters are deleted, likewise; the order of actual parameters is adapted to the new order of the permuted formal parameters.

This is again made possible by the fact that formal parameters of functions, like all other Lava entities, have unique internal identifiers and actual parameters refer to these internally.

3 Facilitating Program Comprehension

3.1 Synoptic Declaration Trees

In section 2.1 we have outlined the nested, tree-like structure of Lava declarations and the associated structure editor. This way to deal with declarations offers decisive advantages for program comprehension:

- You need not put all declarations on a single level but you can nest them according to their primary or subordinate, auxiliary nature.
- You can expand and collapse entire subtrees and in this way switch easily between nested details and "bird's eye view", just as you need.
- You can easily navigate forth and back between declarations and references by clicking the tool buttons "go to declaration" (or double-clicking the reference) and "return to reference".
- You can easily re-arrange the tree structure by applying drag-and-drop or cut/copy/paste operations to individual tree items or to entire subtrees.

As for the “static” nested classes of Java, nesting of declarations does *not* establish a special semantic relationship between inner and outer declarations but is *only* a means to arrange primary and auxiliary declarations in a meaningful way. Inner declarations can always be referenced also from outside, unless they are nested in an *implementation*, or in a package that has been declared *opaque* explicitly.

3.2 Earlier and More Complete Error Reporting

Lava has no compilation phase but checks for errors after every individual structure editing operation. So errors are detected and reported "in embryo", and errors in executable code are highlighted by displaying the erroneous construct (mostly a single identifier or constant, rather than just an entire line of code) in boldface and red color. Likewise, placeholders that have not yet been replaced with concrete constructs in executable code are displayed in red font.

Erroneous declarations are highlighted by a small red rectangle that is affixed to the right side of the corresponding declaration icon. Error messages belonging to the current selection are displayed in a separate error window (for declarations as well as for executable code).

So you have just to look for remaining red flags in declarations and for red portions of executable code in order to figure out where your program calls for correction of errors or for completion. To this end, you can move the current selection to the next or preceding error in the declaration tree view as well as in the exec view.

Moreover, comprehension of still incomplete and erroneous programs is greatly facilitated in Lava by providing several additional features that allow us to perform more complete checks for semantic errors:

1. "Single-assignment" (section 3.5) prevents inadvertent reuse of the same variable within the same program branch with different meanings; violations are reported as errors; the last preceding conflicting assignment is highlighted on a button click.
2. Single-assignment, combined with a stringent phase-model of object creation, initialization, customization and use enables complete initialization checks. Incompletely initialized objects may be passed as parameters only to "initializers" (corresponding to constructors in Java/C++); they cannot be used for method calls. Initializers must initialize all non-optional member variables; violations are reported as errors.
3. The distinction between "value objects", that become immutable after initialization/customization, and "state objects", that may be changed again and again, provides another kind of redundancy, which enables valuable additional checks (cf. section 3.4).
4. Lava supports an advanced notion of "virtual types" with covariant specialization (section 4). This opens a new dimension of static type checking and early error reporting where you otherwise would have to resort to "type casts" and run time type checks in C++ and Java.

3.3 Strict Separation of Interfaces and Implementations

Older, non-object-oriented languages like Modula-2 and the original version of Ada that were based on "abstract data types", had already achieved a clean syntactic separation of "interfaces/definitions" and "implementations" according to the important "principle of information hiding", which we deem to be of vital importance for program comprehension and software maintainability / evolvability.

This clear separation has been lost again in object-oriented languages like C++ and Java. Although Java provides an interface notion, while Java classes contain the implementations of their member functions, you can still use classes to declare the types of variables. A Java interface does not have member variables but only member functions and thus is not suited for specifying a data structure together with a collection of methods that can be applied to it.

In contrast to this, Lava interfaces may contain member functions *and* member variables, and they are the *only* means to declare the types of variables.

Unlike Java classes, Lava implementations implement exactly one interface (and thus have the same name as the interface), They serve *only* for implementing their corresponding interface and they do not inherit from other implementations. They may contain private member variables and functions; these are not exposed to the outside world by the corresponding interface.

An interface may be marked as being "creatable". Then it may be used in "new" operations to specify the type of the objects to be created. It is the job of the Lava run time system to find an implementation of a given interface, as well as implementations of all direct and indirect base interfaces. On creation, a Lava object is composed from all these inherited interfaces and the associated implementations. Lava supports multiple inheritance with "virtual base classes", as you would say in C++: If a Lava interface A inherits the same base interface B several times on several inheritance paths then an object of type A contains only one base object of type B. See Fig.1 (last page) for an example involving interfaces, implementations, and multiple inheritance.

3.4 Distinction Between State and Value Objects

One of the most unusual (and experimental) features of Lava is its distinction between "value objects", that become immutable after initialization/customization, and "state objects", that may be changed again and again. This requires some additional consideration to be invested by the programmers but we believe that this extra cost will pay off later (during program maintenance) by increased comprehensibility of the program.

It is just a big help in understanding the purpose and role of a variable if we know that it does not represent a variable state that can be changed again and again, but just a complex value (therefore "value object") that is constructed and completed once and that is never changed thereafter during the run time of the application. Moreover, as we have pointed out already in section 3.3, point 3, this distinction enables additional, valuable semantic checks.

3.5 Data Flow, Globals, Single-Assignment

Lava prevents all kinds of implicit data flow through global variables or static member variables by relinquishing these concepts and by allowing only explicit data flow through parameter passing and local variables.

Single-assignment, applied to parameters and local variables occurring in the same exec, makes sure that those variables cannot be reused in different meanings within the same branch of this exec. (See section 2.1 for an explanation of the "exec" notion.) Single-assignment has far reaching consequences. It enforces, for instance, a more standardized and regular way to deal with branching constructs:

```
set b ← true;
...
if ...
then set b ← false // error: b has already been set
#if
```

violates the single-assignment rule and would have to be replaced by

```
if ...
then set b ← false  // OK
else set b ← true   // OK
#if
```

Single-assignment excludes also traditional sequential loops that forward information from one pass of the loop to the next by explicitly assigning new values to certain variables in every pass. In Lava, the role of traditional loop constructs is taken over by logical quantifiers "exists" and "foreach" running over finite sets of objects, and by recursive functions. Existential and universal quantifiers replace search loops and exhaustive loops, respectively, whose passes are independent of each other and could be executed concurrently therefore. All other kinds of repetitive algorithms are expressed by recursive functions.

This shift of perspective away from multiple assignment and loop constructs towards a more mathematical view of algorithms, based on exclusive logical distinctions, quantifiers and recursive functions will certainly require some relearning. But it promises to lead to smaller, more self-contained functions eventually and it will greatly facilitate program comprehension if programmers learn to think in these terms.

Absence of global variables and single-assignment cause the data flow to be strictly directed from top to bottom within executable code: The data flow of programs is clarified in a similar way as the control flow has been clarified by abandoning "go to".

4 Design Patterns and Genericity

Lava allows declarations to be grouped in "packages" similar to Java packages. Lava packages are contained completely in one Lava file and are just a special type of nodes in the Lava declaration tree. Packages and interfaces may be endowed with type parameters, called "virtual types". These may be overridden in derived interfaces and packages by assigning more derived types to them. The types of member variables and method parameters may be such virtual types. Based on this virtual type notion, Lava allows you to define groups of mutually recursive interfaces with "covariant specialization" of (virtual) member and method parameter types. This is a very natural way to support reusable "design patterns" in the sense of [2] (cf. also [6], [12], [13]) and a powerful alternative to the traditional parametric types/templates of C++ [11], Eiffel [9], and the Java genericity extension GJ [5]. Another highly desirable consequence of using patterns is that "covariant specialization" renders the ubiquitous "type casts" of C++ and Java programs superfluous.

The extension of the derivation and (multiple) inheritance notions from interfaces to patterns/packages, combined with declaration nesting, is perhaps also an appropriate way to describe the derivation / descendence relations between the individual patterns of a "pattern language" [6] or at least certain aspects thereof.

5 Conclusion

Quite a number of unusual features establish the novel and experimental nature of Lava. The use of structure editors instead of text editors relieves the programmers from syntax learning and prevents syntax errors from the beginning. The synoptic tree representation of nested declarations with its collapse/expand, drag-and-drop, go-to-declaration, override support and other functions will greatly facilitate program (re)structuring and program comprehension. The distinction between immutable value and variable state objects allows us to express more semantics in Lava. The more stringent object initialization/customization discipline, the single-assignment concept, the absence of global variables and traditional sequential loop constructs will clarify the data flow and enforce more standardized program structures based on small recursive functions. Some of these features enable more detailed semantic checks and early error reporting. Advanced support of genericity by "virtual types" opens a new dimension of program structuring by reusable design patterns. It avoids ugly "type casts" and enables a higher amount of static type checking.

Although not treated in this paper: The purely declarative treatment of concurrency, synchronization and transactions and the seamlessly integrated support for database queries, based on logical conjunctions, quantifiers and a "select" expression (as a substitute for "embedded SQL") promise to greatly reduce the learning effort of database programmers and to remove the root of many potential errors.

References

1. Caprile, B., Tonella, P.: Restructuring Program Identifier Names. Proceedings IEEE ICSM'00, 2000, ISBN 0-7695-0753-0
2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995, ISBN 0201633612
3. Gosling, J., Joy, B., and Steele, G., Bracha, G.: The Java Language Specification. Addison-Wesley, 2000, 896 pages, ISBN 0201310082
4. Günther, Klaus D.: Lava – Programmieren im Lego-Stil. Proceedings Component Developers and Users Forum 2001
5. GJ: http://www.cs.bell-labs.com/who/wadler/pizza/gj/Documents/index.html
6. Hillside Group, Pattern Home Page: http://hillside.net/patterns/
7. Java: http://www.javasoft.com
8. Lava: http://www.darmstadt.gmd.de/~guenthk/Lava/
9. Meyer, B.: Eiffel: The Language. Prentice Hall Europe, 1992, ISBN 0132479257
10. PITAC Report to the American Government: http://www.ccic.gov/ac/report/
11. Stroustrup, B.: The C++ Programming Language, Special Edition. Addison-Wesley (2000), ISBN 020170073
12. Thorup, K.K., Torgersen, M.: Unifying Genericity (Combining the Benefits of Virtual Types and Parameterized Classes). Proceedings ECOOP'99, 186-204
13. Tonella, P., Antoniol, G.: Object-Oriented Design Pattern Inference. Proceedings IEEE ICSM'99, 1999, ISBN 0-7695-0016-1

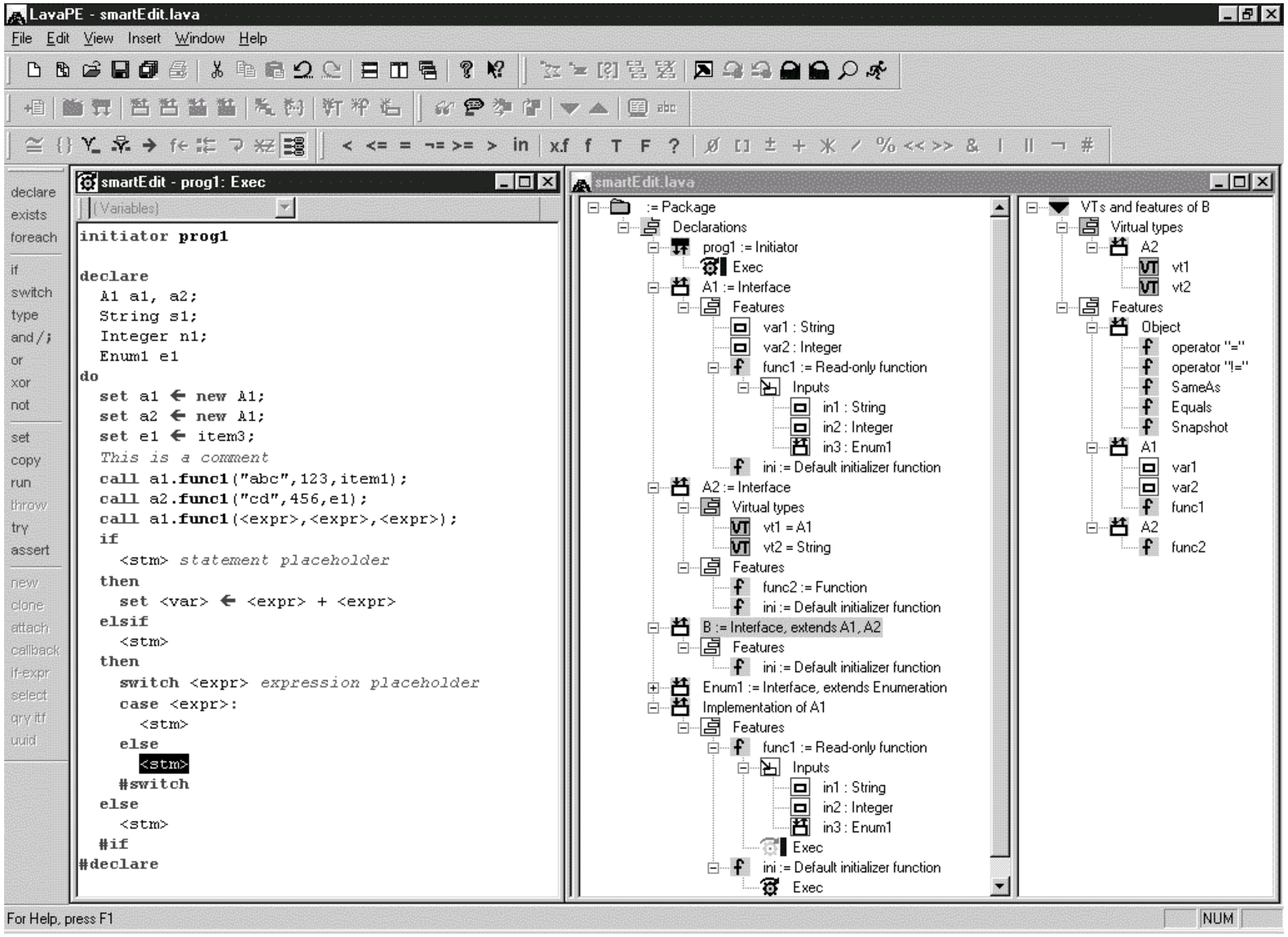


Fig. 1: Lava exec, declaration, and override view

Feature Modeling

Andreas Speck[1] and Elke Pulvermüller[2]

[1] Intershop Research Jena
D-07740 Jena, Germany
a.speck@intershop.com
[2] Institut für Programmstrukturen und Datenorganisation,
Universitaet Karlsruhe,
D-76128 Karlsruhe, Germany
pulvermueller@acm.org

Abstract. Component-based approaches for the software development are well-known. Most of these approaches (like CORBA and COM++) focus on the realization of the connection between the components and their interactions.
In this paper we concentrate on the missing items in the component-based software engineering: the problem how to model component composition and to validate them. We apply the component model based on the interface description with *InPorts* and *OutPorts* which allow a rather detailed definition of the components interaction. Moreover we take the term feature to name the core requirements to a component.
Features are used to drive the description of the component composition which is regarded as an combination of features expressed by logical operators. Moreover the *InPorts* and *OutPorts* describe the dynamic component interactions. The combination of *OutPorts* and *InPorts* according to the component composition rules allows to reason about the component system's dynamic behavior.

1 Introduction

The software development from components is increasingly gaining importance. Like with objects there exist different definitions for components The most well-known is given in [8]. Components and their composition are supported by various technologies such as CORBA, COM/DCOM or Java Beans. Components themself are already beyond being issue of research only. Different commercial vendors provide components as building blocks (e.g. Enterprise Beans) for industrial systems.

Now the question rises how to build up (or model, design and verify) interactions between the composed components. We consider that a component provides one feature or a set of features. The design of a system of components is driven by the desired features and their interactions [3], [5].

1.1 Component Models

There exist various component models. Most interesting are models that focus on interfaces of the components. In [4] *InPorts* and *OutPorts* are used to

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

specify these interfaces. The protocols of the *InPort* and *OutPort* interactions are specified by automates. Such a component is depicted in figure 1.

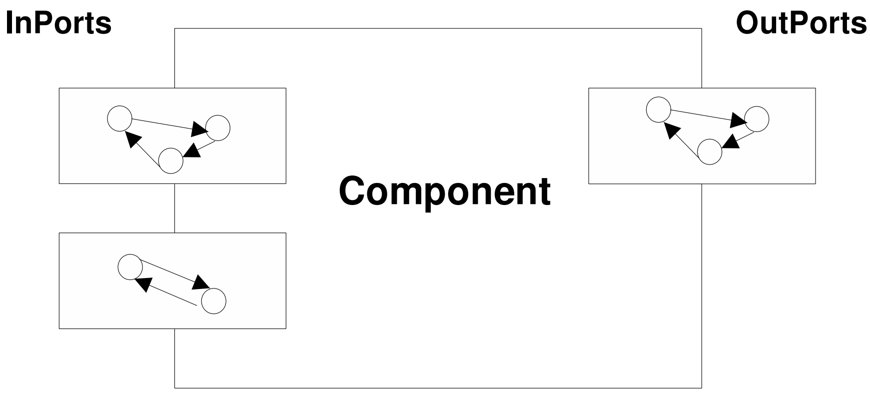


Fig. 1. Component Model with *InPorts* and *OutPorts*

The combination of components is described by the combination of the specific *InPort* and *OutPort* automates. These automates may be the base for an automatic verification of the interfaces between the components. This approach is rather old. The concept on in- and out-interfaces and the verification of their equivalence was already an issue in research area of modules [1].

1.2 Features

Features are properties of components. A component may provide a single feature or a set of multiple features. A feature itself may consist of other features.

Features are either basic features (usually representing a single method or attribute) or combined features (c.f. section 2).

The features or respectively the names of the features have to be annotated to the objects, methods or attributes of the components. In the most simple case this has to be done manually by the developer of the component as documentation of the feature. Composed features are expressed by their sub-features.

2 Feature driven Composition

Features are properties of components. In the same way as components may be combined in new larger components features may be combined too. Features may be combined form other (sub-)features. E.g. the feature *text editor* consists of several features:

- *editing algorithms* like *insert*, *undo*, *save* and *load*
- *text attributes* e.g. text which itself may contain other attributes (or features) like *words*, *sentences* or *paragraphs*

The feature *text editor* may be expressed as:

$$\begin{aligned} \textit{text editor} &= \textit{editing algorithms} \wedge \textit{text attributes} \\ \textit{editing algorithms} &= \textit{insert} \wedge \textit{undo} \wedge \textit{save} \wedge \textit{load} \\ \textit{text attributes} &= \textit{words} \wedge \textit{sentences} \wedge \textit{paragraphs} \end{aligned}$$

Of course all basic logical operations are available in order to combine features. Figure 2 shows the notation which may be applied. In the example the *SuperFeature* contains the *SubFeature A*, *SubFeature B* and *SubFeature C*:

$$SuperFeature = SubFeature\ A \wedge SubFeature\ B \wedge SubFeature\ C$$

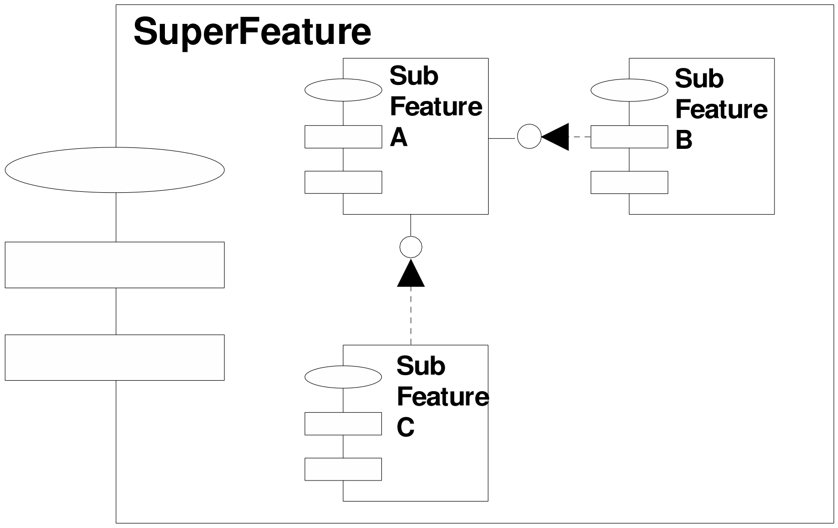


Fig. 2. Feature-driven Combination

3 Feature Validation

The application of a logic description of feature combinations enables to control and verify systems. A system may be designed as logical dependencies of different features. Such a logic design can automatically be processed and may serve as base for automatic combination tools (generators [7]).

In general there are two potential ways of verifying or validating features of a system:

1. Static Composition
   The static design may be verified by adding rules, e.g. *Feature A* requires *Feature B*, or *Feature A* excludes *Feature C*. When a specific design has to be reused the appropriate rules may simply be reused. In order to improve reuse the rules may be stored in a database.
   A more detailed description of this type of verification may be found in [3].

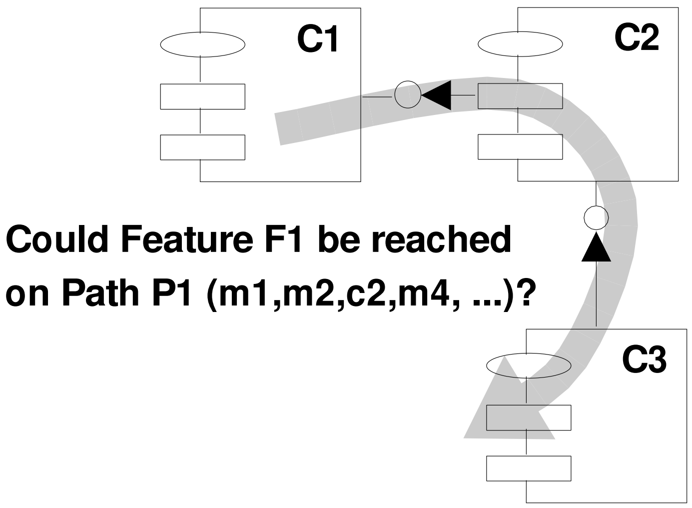


Fig. 3. Reasoning about Features Combination

2. Dynamic Behavior
   In addition to the static composition it may be interesting to reason about the dynamic interaction of a system. Figure 3 presents an example: Could a specific feature be reached when a defined sequence of input messages is given to the system?

   Other questions may be:
   - In which order are specific features passed when a message is sent to the system?
   - When there are specific alternatives in a feature system, which feature is passed when a specific condition is given by a specific message?
   - Does the system run into a dead-lock?
   - Do redundant paths exist in a system?

The validation of the dynamic behavior may be supported by tools. An approach may be the combination of the *InPort* and *OutPort* automates by additional automates representing the internal activities of a component. Component automates defined in this way may be combined in order to specify the systems behavior. A detailed description of this approach may be found in [6]. Moreover in this approach missing methods of components may be detected. Based on this an automatic adaption of the component system is possible by adding adaptors providing the missing methods.

Since it is hard to determine state charts describing the internal behavior of a component automatically we decided to parse out UML sequence charts [9]. Moreover the sequence charts may be reduced to the sequences of direct impact from the *InPort* states to the *OutPort* states.

In our approach we applied a model checking tool [2] to validate the system's dynamic interactions (c.f. figure 4). Therefore we define an *Actor* which sends

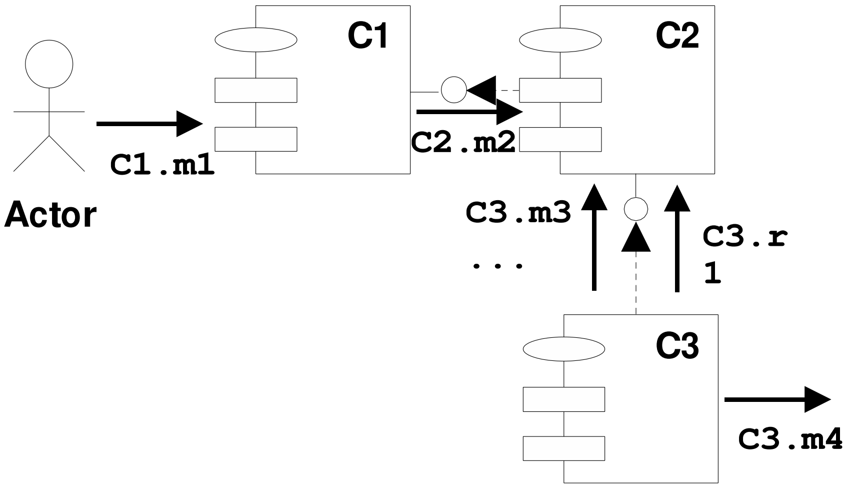


Fig. 4. Actor triggering Components Interactions

messages to the system to be validated. Within the system we determine the dependencies and communication relationships between the components. Now the model checking tool allows to reason whether a specific message or feature is triggered by message given by the *Actor*.

4 Conclusion and Future Work

In this paper we present a method to describe the design of a component system by the logical combination of features. We introduced the term feature in order to define the basic intentions of components which may be realized by methods, data or objects. The combination of this features drive the components composition.

Besides this statical view of a component system we consider the dynamic behavior as well. We consider the components' interfaces (*InPorts* and *OutPorts*) as a key issue for the specification of the dynamic interaction between components. *InPorts* and *OutPorts* are defined by automates representing their temporal actions. Within a component both may be connected by additional automates or sequence charts in order to define the components' behavior. The combination of *OutPorts* and *InPorts* of dependent components allows to reason about their interactions. In this way entire component systems may be validated automatically.

An interesting issue of our future work will be the feature composition guided by artificial intelligence. This is an improvement of the feature design data base which may help to partially reuse existing design or to detect critical combinations which do not necessarily lead to errors.

The application of AI may also help to extend the component combination approach considering not only the boolean operations conjunction, disjunction and negation but also fuzzy values in-between the range of true and false.

References

1. P. Gouthier and S. Pont. *Designing Systems Programs.* Prentice Hall, Englewood Clifs, 1970.
2. McMillan K. *Symbolic Model Checking.* Carnegie Mellon University, 1992.
3. H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect Composition applying the Design by Contract Principle. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering GCSE'2000*, Erfurt, Germany, October 2000.
4. A. Lauder and S. Kent. EventPorts: Preventing Legacy Componentware. In *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC 99)*, 1999.
5. E. Pulvermüller, A. Speck, and J. Coplien. A Version Model for Aspect Dependency Management. In *Proceedings of the Third International Symposium on Generative and Component-Based Software Engineering GCSE'2001*, Erfurt, Germany, September 2001.
6. R. Reussner. Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten. In *Proceedings of the GI joint Workshop of GI AK 2.1.9 and AK 2.1.4*, Bad Honnef, Germany, May 2001.
7. Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Lecture Notes in Computer Science LNCS 1445*, pages 550 – 570, 1998.
8. C. Szyperski. *Component Software.* Addison-Wesley, ACM-Press, New York, 1997.
9. W. Vanderperren and B. Wydaeghe. Towards a new Component Composition Process. Brussel, Belgium, 2001. Vrije Universiteit.

Adapting Components and Predicting Architectural Properties with Parameterised Contracts

Ralf H. Reussner

Universität Karlsruhe (TH), Am Fasanengarten 5, D-76128 Karlsruhe, Germany
reussner@ira.uka.de

Abstract. [1] While interoperability tests check the inclusion of a component's requires interface in the environmental provides interfaces, parameterised contracts link the provides- and the requires-interface of a component. This allows to perform automatically a certain class of component adaptations without changing the code. These adaptations also do not have to be foreseen and programmed by the component developer in advance. We show how parameterised contracts can be used to enhance the reusability of software components in software architectures. Combining parameterised contracts themselves results again in a parameterised contract, describing the properties of a component assembly. We present to common connection styles, which can be used to conclude from local component properties to global architectural properties.

1 Introduction

In component oriented programming (COP) software components are regarded as units of late composition [17]. This view is justified by the anticipated benefits of reusing software components (considering time-to-market, costs, and quality of software). Due to that view, a lot of technical-oriented research in component-oriented programming was concerned with composition-time interoperability checks (e.g., [4, 6, 18]). Several new interface models have been proposed which allow to detect more interoperability errors than the currently used signature-list-based interfaces do.

Most work on software architectures (e.g., [7, 16]) looks at structural interactions (i.e., connectors) between components and abstracts away from concrete interface models. More recent work in software architecture (like [5]), also consider dynamic reconfigurations of composed systems, or take a more process-oriented view to apply software architectures to create product families [1]. Most architectural description languages (ADLs) model for components the explicit separation of the provides-interface (i.e., the services offered by a component) and the requires-interface (i.e., the external services required by a component) and intentionally do not restrict themselves to a certain interface model.

[1] This paper is an extension of [14].

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

Bringing together software architecture research and component oriented programming can have several benefits:

- a higher degree of reuse of existing components in new software architectures
- better support of dynamically reconfigurable software architectures
- (in the long term) a better engineering foundation of architecting software systems, if we understand how local component properties interact with global system properties.

To yield these benefits, we have to ease the application of components not only as a unit of late composition, but also as a unit of architectural composition. This support of component integration relates to a hierarchy of research problems.

1. When to accept / reject a component for integration into a new architecture. This question relates to well-known problems of interoperability research: (1) How to ensure the correct usage of a component by the rest of the system (i.e., other components, a framework, etc.), and (2) how to ensure that all resources required by a component are available?
2. How can we adapt a component in case it does not fit exactly into the environment? On the one hand, practical experience shows that existing components usually cannot be reused in a new context as they are. On the other hand, users of components cannot or are not willing to change components. So what we need is a adaptation support by the infrastructure.
3. How to conclude from local component properties to global system properties. (Recently one concentrates on non-functional properties or quality attributes.)

This paper concentrates on question two and tackles question three. The contribution of this paper lies in the formulation of parameterised contracts and in demonstrating their relevance for software architectural problems. Parameterised contracts [11] are a generalisation of classical contracts, as defined by Meyer [8]. We use parameterised contracts to restrict a component's provides- and requires-interface, hence adapting the component to specific reuse contexts. Not touching the code allows us to apply this concept to black-box-components. Implemented in the infrastructure of a component system, neither the component developer nor the system developer has to foresee and manually program these kind of adaptations. This reduces component and system development costs.

In section 2 we further discuss the links between COP and software architecture, motivate the importance of parameterised contracts and show their relevance for component design and system architecture. In section 3 we present parameterised contracts as an approach for component adaptation. In section 4 we show the application of parameterised contracts with examples of two different interface models. In section 5 we show how we can compute a parameterised contract for simple forms of component assemblies (pipelined and bundled components). There we present two ways to combine parameterised contracts (corresponding to pipelining and bundling) and mention some inherent properties of these combinations. These combinations of parameterised contracts can be used to predict the properties of the component assembly out of the component's properties. Section 6 concludes and discussed the application of parameterised contracts as a concept to conclude from local component properties to global system properties in general (question three).

2 Linkage between Component Design and System Architecture

2.1 Use of Components and Generators in Software Architecture

Interface information of a component can be used to decide whether to reuse a component or not. Hence, we can utilise interface information for the important issue of software architecture evaluation in early stages of development. When to decide between two alternative software architectures, we can choose the architecture, which achieves a higher degree of reuse. This degree of reuse strongly depends on the effort one has to spend to adapt existing components into the new architecture's context.

Without concrete interface information of a component, we usually cannot estimate the effort of the component's adaptation. Generally, we see two ways how CASE tools can use interface information of software components to ease the adaption and to estimate the effort of adaption: (a) Semi-automatic generation of adaptors out of interface information (e.g., [15, 20]). The more complex the adaptor is, the higher the effort of integration (and the likeliness of programming errors). (b) The automatic adaptation of a component by a generator. In case the result of that adaptation suffices the requirements, the component is integratable. In case of not, we can try to generate adaptors. Hence, the automatic adaptation tries to combine a component oriented approach with an generator-oriented approach to achieve a higher degree of reuse of components (generation of adapted components and generation of adaptors). In our context the deployment of generators during system design is sufficient, but also the deployment of generators for composition-time adaptation has its applications.

The class of adaptations which can be performed with parameterised contracts is the class, where components restrict their requires-interface (in case the component itself has not to offer all functionality), and vice versa, where components restricts its provides-interface (in case the environment offers not all requested functionality).

In the following we motivate why this class of adaptation is practically relevant and relates to the design of components *and* system architectures.

2.2 Component Design and Reuse in Software Architectures

The granularity of a component is an issue of component design, which has a strong effect on the component's reusability. On the one hand, large grained components are justified by the observation that most users of a computer program only use a subset of the program's functionality. Hence, translated to components, this means the more functionality a component offers (i.e. the larger grained the component is), the more users will find their requested functionality as a subset of the component's functionality and will reuse the component.

On the other hand, adding new functionality does not improve a component's reusability in general, since the added functions translate into new requirements to the environment where the component is to be embedded. Thus, the component becomes less reusable, contrary to the original intention. This we call the *'granularity-reuse problem'*. For example, imagine you are designing a printer management component. If you restrict its functionality to handle only local printers, it will not be very reusable because it will not handle network printers. However, if you design the component for network printers, it will require a network even for managing the local printer, and that will not make it very popular with users.

So the component's granularity (which is an issue of component design) has a strong influence on system design, since the system has to provide all functionality the component requires. Components of a large granularity usually require more external components than components of a small granularity. But building only fine-grained components is no solution. Although they have less strong external dependencies, they introduce many unnecessary interfaces into the system (which degrades its performance). Furtheron, they are very specific, so less reusable and less attractive for component manufacturers than larger-grained components.

The granularity-reuse-problem arises because components have static provides- and require-interfaces. In all future reuse contexts the component will offer all functionality, which was specified in its provides-interface at design time. Regardless, whether this functionality is completely used or not, also the component will always require all functionality, which is specified in its requires-interface. Hence, what we need, is a mapping of the functionality, which is *actually* requested from a component by a system, to the functionality, which the component *really* requires, to provide this requested functionality. For reenginering existing systems, the reverse mapping is also useful (i.e., adapting new components to existing systems). These kinds of mappings are provided by parameterised contracts.

3 Parameterised Contracts

3.1 Interoperability Checks and Classical Contracts

Interoperability checks between components relate to "classical" contracts for software components. A contract for a component specifies, under which precondition a component A has to fulfil the postcondition. Translated to interoperability checks, we check if the environment (represented by component B) offers all functionality A expects. Thus we check the inclusion of A's requires-interface (precondition) in B's provides interface (see point (1) in figure 1). Is this check successful, A fits into the environment and will offer all services of its provides-interface (postcondition). Hence, interoperability checks have a boolean outcome: a component fits into system or not. But for practical reasons the more interesting and relevant question is, what can we do, if a component does not fit into its environment.

3.2 Parameterised Contracts as Generalisation of Classical Contracts

While interoperability tests check the requires-interface of a component against the provides-interface of *another* component, parameterised contracts link the provides-interface of one component to the requires- interface of *the same* component (see points (2) and (3) in figure 1). Classical contracts [8] do not reflect this connection between provides- and requires-interfaces. Classical contracts, once formulated statically during component development, cannot change the post- or precondition according to a new reuse context. This motivates the formulation

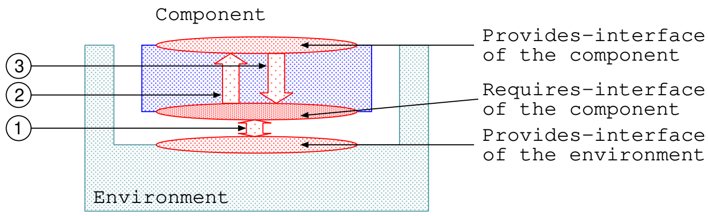


Fig. 1. Interoperability checks (1) and Requires-parameterised Contract (2) and Provides-parameterised Contract (3)

of two kinds of parameterised contracts.

- provides-parameterised contracts map the provides-interface to a requires-interface.
- requires-parameterised contracts map the requires-interface to a provides-interface.

Technically spoken, parameterised contracts are a mapping which is bundled with the component and computes the interfaces of the components on demand. The requires-parameterised contract takes as arguments the requires-interface of the component and the provides-interface of the environment. Hence, parametersised contracts are isomorphic mappings between the domain of preconditions and the domain of postconditions. Both domains can be modelled as partially ordered sets (posets). (A mathematical discussion of parameterised contracts can be found in [13].) The intersection of a components requires-interface and the enivoronment prevides-interface describes the functionality which is requires by the component and provided by the environment. Out of that information the requires-parameterised contract computes the new provides-interface of the component. Analogously, a proviodes-interface computes the new requires-interface out of the privides interface of the component and the requires-interfaces of its clients. In our prototypical system [13] we implement parameterised contracts within the component's reflection system.

The following scenarios demonstrate the application of parameterised contracts. Provides-parameterised contracts are useful when designing a new system. A software architect states the functionality a component has to fulfil. Then she finds several candidate components in a repository, which deliver at least the required functionality. For all these candidate components one can compute the functionality they really need in this context via their provides-parameterised contracts. This can be done because the functionality the component has to provide in this context is known.

Requires-parameterised contracts are useful for integrating components into existing systems. Imagine an existing system should be reconfigured with a new component, or an existing system architecture should be enhanced by an existing component. One question in this situation is which functionality the new component will deliver without changing the environment of the component. For creating fault-tolerant system architectures requires-parameterised contracts can be applied at run-time. This is useful because in fault-tolerant system it is important to know, which functionality a component can offer when other components fail.

Parameterised contracts are an abstract concept, which is not tied to a specific interface model. Like most ADLs, parameterised contracts only need the explicit separation of provides- and requires-interfaces. Furtheron, parameterised contracts can be applied at design-time (as strongly proposed in this paper), but also during composition-time (e.g., to integrate components into existing systems) or at run-time (e.g., for graceful system degration). Like classical contracts, parameterised contracts can be applied for a number of different software units, such as methods, modules, objects and components. (But parameterised contracts may prove most useful when applying to components.)

Applying parameterised contracts to software components means, that the interfaces of the component are recomputed dynamically. The code has not to be manipulated. For practical reasons is is important, that the programmer does not have to foresee and program all possible component adaptations in advance, which are computable by parameterised contracts.

More generally, parameterised contracts reflect the fact, that the properties of a component cannot be regarded in reality as absolutely fixed attributes of the component. Much more, component properties often strongly depend on the specific context, the component is deployed in. For example, the timing behaviour of a component cannot be given by some fixed numbers of miliseconds, because it depends very muchon the timing behaviour and guarantees of the underlying environment (middleware, OS, hardware). Similarily, the reliability of a component depends on the reliability of its environment.

For components with strongly connected provided and required services parameterised contracts may not make sense. E.g., for a screen-saver without access to a graphic-device a requires-parameterised contract cannot compute a meaningful provides-interface.

4 Example

As mentioned, parameterised contracts are not tied to a specific interface model. Here we will show the application of parameterised contracts for a simple interface model (signature lists) and a more sophisticated one (including protocol information, modelled with finite state machines).

As an example, regard a multi-media video-mail component. This mail not only contains the video itself, but also offer functionality to present the video. This design is useful, if you want to abstract away from specific video file formats or if you want to handle different media (like text, sound, and video) in the same manner. The `videoMail` component makes use of two other (system-specific) components: `videoPlayer` and `soundPlayer`. The functionality offered by `video-player` contains in its provides-interface the methods `start`, `stop`, `pause`, `volumeUp`, `volumeDown`, and certain methods to adjust the picture, like `brightnessUp`, `brightnessDown`, etc. In case `videoMail` arrives on a system without sound support (e.g., due to hardware reasons) a require-parameterised contract computes a restricted provides-interface without the methods `volumeUp` and `volumeDown`.

Additionally to signature-lists, it is often valuable to include more information in components' interfaces, such as a description of valid call sequences to the component's services [6, 9]. This description of supported call sequences for the provides protocol of a component. The unrestricted provides protocol of `videoMail` (figure 2, left) models the allowed sequences of method calls to `videoMail` with a finite state machine. In practice, the set of allowed call se-

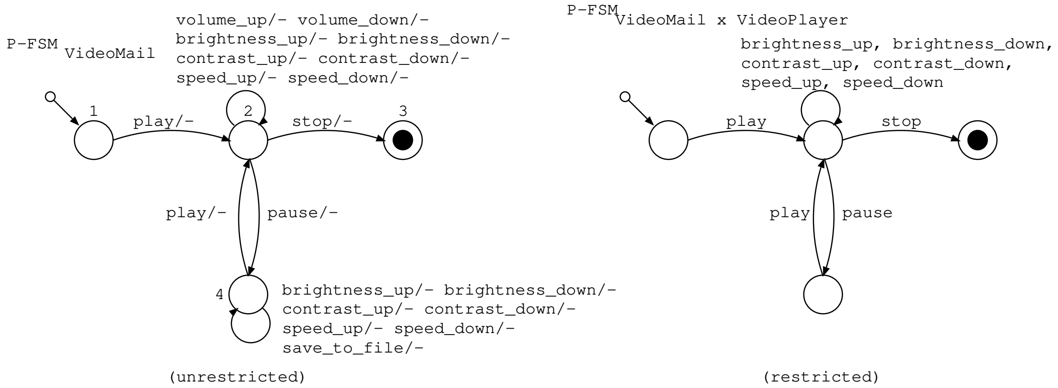


Fig. 2. The provides protocol of the `videoMail` component. Unrestricted version (left), restriction to specific reuse context by require-parameterised contract (right)

quences depends on the specific reuse context (i.e., the call sequences the underlying `videoPlayer` supports). So one can imagine that a requires-parameterised contract restricts the functionality of `videoMail` in a specific reuse context to the protocol shown with the finite state machine in figure 2 (right). Note that modeling the protocol reveals functionality changes which are not expressible with signature lists, such as the availability of method `constrastUp`.

The technical realisation of parameterised contracts including the linkage between interfaces is described in [12] for an interface model with finite state machines. [13] extends this algorithms to an interface model of greater practical relevance which also can model common container classes like stacks, which cannot be modelled exactly with finite state machines.

5 Combining Parameterised Contracts to Predict Architectural Properties

To revisit question three of the introduction (how to conclude from local component properties to global system properties), we can combine parameterised contracts to reflect the combination (assembly) of components. As argued before, parameterised contracts are well-suited to describe properties (functional and non-functional) of components, since a parameterised contract makes the existing relations between provides- and require interface explicit.

When regarding the composition of components as a component itself (and the benefits of hierarchical decomposition during design suggest us to do so), we would like to describe the composition of components with a parameterised contract again. Hence, the question is, how to combine parameterised contracts of components in a way, that the combination of parameterised contracts itself is again a parameterised contract which describes the assembly of these components. Therefore we have to consider, that the combination of parameterised contracts must reflect the way the components interact (i.e., the architectural pattern: e.g., pipe-and-filter [3, 16], master-worker, clients-and-server, etc.).

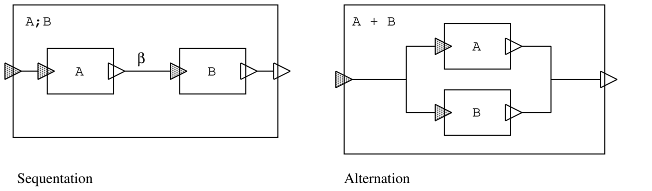


Fig. 3. Combinations of parameterised contracts: sequentiation (left) and alternation (right)

Two common ways to combine components are (a) pipelining of components and (b) bundling components (meaning that several components together provide their services directly to clients, for example behind a facade). For both combinations of components corresponding combinations of parameterised contracts exist. For pipelined components we combine parameterised contracts by sequentiation (figure 3 (left)), for bundled components parameterised contracts are combined by alternation (figure 3 (right)). Both kinds of combinations of parameterised contracts have their specific properties: The provides interface of the sequentiated parameterised contract will never be stronger than the provides interface of component A. But it can be restricted if B's provides interface does not include A's require interface completely. Analogously, the requires interface of the sequentiated component will never be stronger than the requires interface of component B. But it can be weaker, if component A requires only a true subset of B's requires interface. In contrast, an alternated parameterised contract is the sum of the parameterised contracts of the inner components, hence it enhances the functionality of the inner component's provides interfaces and requires interfaces. Therefore it can be used the describe the effects of plug-ins. A mathematical treatment of these intuitive arguments is given in [13].

Sequentiation and alternation are two very simple forms of component assemblies with respect to the combination of parameterised contracts. When looking at concurrent usage of one component (a server) by several other components (clients), issues of synchronisation must be considered when predicting properties of these component assemblies. In some cases synchronisation can be handled by the architectural style, in other cases specific generated connectors can ensure certain synchronisation properties (like fairness or absence of deadlocks) [15].

6 Conclusions

We presented parameterised contracts as a generalisation of interoperability checks between components (i.e., classical contracts of components). Applying parameterised contracts to software components allows to perform adaptations of components automatically, which restrict the component's provides- or requires-interface. The base of parameterised contracts lies in the observation that in most practical cases the provides-interface and the requires-interface of a component are not isolated: A component will offer less functionality if its environment offers not all functionality the component requires (to provide all programmed functionality). And, a component will require less functionality, if not all offered functionality of the component is to be used by its clients. Hence, it enhances the component's reusability to compute the component's provides-interface out of its required-interface and vice versa.

The linkage between a component's provides- and requires-interface is independent from the concrete interface model. So we can apply parameterised contracts to simple signature-list-based interfaces, but also to protocol-based interfaces, like demonstrated in the examples.

Future work will be concerned with models of more complicated combinations of parameterised contracts beyond sequentation and alternation. This includes communication patterns like publisher-subscriber and others. To conclude from local component properties to global architectural properties two issues are of concern: (a) the inclusion of the properties to reason about in the interface model of a component, and (b) the proove of certain additional global system properties. For example, for talking about the timing behaviour of an architecture it is not sufficient to reason about the timing of the single components, but also it must be sure, that no deadlocks of the component interaction destroy any timing properties of the system. Here transactional systems (e.g., [19]) and other modelling techniques, appropriate for global concurrency analysis (such as Büchi-Automata [2] or Petri-Nets (e.g., [10])), may be useful.

References

1. Jan Bosch. *Design and Use of Software Architectures – Adopting and evolving a product-line approach.* Addison-Wesley, Reading, MA, USA, 2000.
2. J. R. Buechi. On a decision problem in restricted second order arithmetic. In E. Nagel et al., editor, *Proceedings of the second international conference on logic, methodology and philosophy of science*, pages 1–11. Stanford University Press, 1960.
3. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns.* Wiley & Sons, New York, NY, USA, 1996.

4. Jun Han. Temporal logic based specification of component interaction protocols. In *Proccedings of the 2nd Workshop of Object Interoperability at ECOOP 2000*, Cannes, France, June 12.–16. 2000.
5. Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley, Reading, MA, USA, 2000.
6. Bernd Krämer. Synchronization constraints in object interfaces. In Bernd Krämer, Michael P. Papazoglou, and Heinz W. Schnmidt, editors, *Information Systems Interoperability*, pages 111–148. Research Studies Press, Taunton, England, 1998.
7. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989:137–155, 1995.
8. Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.
9. Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15, October 1993.
10. Wolfgang Reisig. *Petrinetze.* Springer-Verlag, Berlin, Germany, 2 edition, 1986.
11. Ralf H. Reussner. Parameterised Contracts for Software-Component Protocols. Presentation given at Oberon Microsystems, Zürich, http://liinwww.ira.uka.de/~reussner/zuerich00.ps.gz, December 2000.
12. Ralf H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *34th Hawaiin International Conference on System Sciences.* IEEE, January 3–5 2001.
13. Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten.* Dissertation, Fakultät für Informatik, Universität Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe, Germany, July 2001. to appear.
14. Ralf H. Reussner. The use of parameterised contracts for architecting systems with software components. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, June 2001. to appear.
15. Heinz W. Schmidt and Ralf H. Reussner. Automatic Component Adaptation By Concurrent State Machine Retrofitting. Technical Report 25/2000, Fakultät für Informatik, Universität Karlsruhe (TH), Am Fasanengarten 5, D-76128 Karlsruhe, Germany, 2000. This report appeared simultaneously as Technical Report No. 2000/81 of the School of Computer Science and Software Engineering, Monash University, Melbourne, Australia.
16. Mary Shaw and David Garlan. *Software Architecture.* Prentice Hall, Englewood Cliffs, NJ, USA, 1996.
17. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press, Addison-Wesley, Reading, MA, USA, 1998.
18. A. Vallecillo, J. Hernández, and J.M. Troya. Object interoperability. In A. Moreira and S. Demeyer, editors, *ECOOP '99 Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag, 1999.
19. Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems – Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann Publishers, San Mateo, CA, USA, 2001.
20. D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

Embedding Processes in a Declarative Programming Language*

– Extended Abstract –

Bernd Braßel[1], Michael Hanus[2], and Frank Steiner[2]

[1] RWTH Aachen, Germany
brassel@halifax.rwth-aachen.de
[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh,fst}@informatik.uni-kiel.de

Abstract. While declarative programming languages are based on the idea of specifying the static relationships of problems, the right modeling of the dynamic behavior is equally important for many practical applications. To support a high-level specification of both aspects of computational systems, we propose the embedding of a process-oriented specification language in a multi-paradigm declarative language. Since this embedding is done in a seamless way, the features of the declarative base language can be exploited (1) for a high-level specification of the computational needs in single state transitions of a dynamic system, and (2) to reuse the abstraction facilities of the base language for the specification of the structure of dynamic systems. We show an implementation of these ideas in the declarative multi-paradigm language Curry. This implementation has been used for a prototypical implementation of embedded and distributed systems in a high-level manner.

1 Introduction

Declarative programming languages (e.g., functional, logic, or functional logic languages) aim to support *high-level* descriptions of software systems, which are *executable* at the same time. Such programming languages have many advantages w.r.t. the efficient development, reliability, maintenance, analysis, and verification of programs. The general idea of declarative programming is the specification of the static relationships of a given problem by well-understood mathematical entities (functions and/or predicates). However, many real-world applications demand also for an appropriate modeling of the dynamic behavior of a complex software system, which may be distributed into communicating active parts or embedded in an environment where they must react on external events.

Processes are an appropriate notion to describe dynamic behavior, and high-level formalisms [3] have been developed to describe the essence of process behaviors, like communication, parallelism, process creation, on an abstract level.

* This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2 and by the DAAD under the PROCOPE programme.

We want to keep the advantages of declarative programming but make them applicable also for a wider range of applications, like distributed or embedded systems. For this purpose, we propose an embedding of a process-oriented specification language in a declarative language, where we choose the declarative multi-paradigm language Curry [4, 9] as our base. On the one hand, our proposal is based on a clear separation between the declarative and dynamic aspects of an application system by making processes a distinguished data type (this has some similarities to the separation of pure functions and functions manipulating the external world by the introduction of monads [11]). On the other hand, our embedding of processes is done in a seamless way by defining processes as a standard data type so that the features of the base language can be exploited in two ways: (1) the computational needs in single state transitions of a dynamic system can be specified in a high-level style, and (2) the abstraction facilities of the base language can be reused for the specification of the structure of dynamic systems.

Our objective is to provide a domain-specific language for the description of dynamic systems that should react on external events (e.g., embedded systems). The integration of such a domain-specific language into an existing high-level programming language has the advantage that we can reuse the functionality of the base language in application programs and we can provide a prototypical implementation of the entire framework with a limited effort. We have used our implementation for a prototypical implementation of embedded and distributed systems in a high-level manner.

This paper is structured as follows. In the next section we provide a basic introduction into our framework. Section 3 sketches the features of Curry as necessary for the understanding of this paper. In Section 4 we show the modeling of our process-oriented specifications in Curry. Its application is demonstrated in Section 5 by several examples before we make some remarks about the implementation of our framework in Section 6 and conclude in Section 7.

2 Specification of Process Systems

Our framework is based on the partition of a dynamic system into several components that cooperate by exchanging messages. In an embedded system, such a component could correspond to a controller that reacts on messages received from external sensors by sending messages to other active components. In a distributed software system, these components may run on different computers and exchange messages via the Internet. Our goal is to provide a framework for the high-level description of such components.

The structure of each component is sketched in Fig. 1. A *component* consists of a set of processes (p1,p2,...), a global state (i.e., data visible for all processes inside a component but not visible from outside) and a mailbox (queue of messages sent to this component). The behavior of a dynamic system is defined by the behavior of each process. A process can be activated depending on conditions on the global state or the mailbox. If a process is activated (e.g., because a

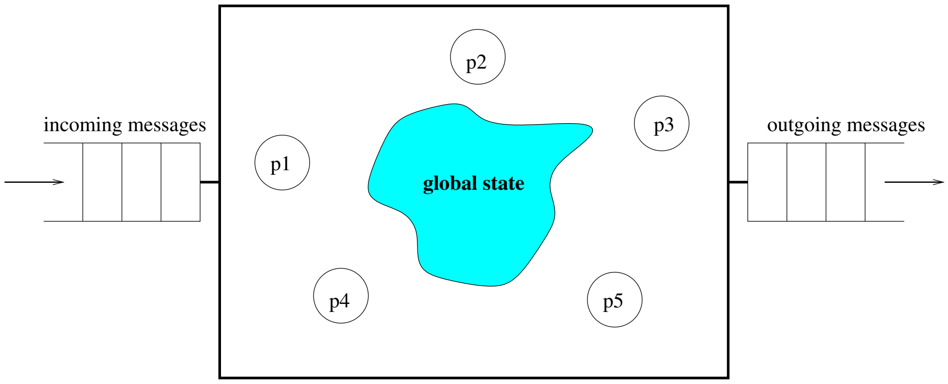


Fig. 1. Component of a dynamic system

particular message arrives in the mailbox), it performs an action and may start other processes (since we have an interleaving semantics, at most one process can perform an action so that actions are atomic entities).

The reaction of a process to the change of its external context (i.e., mailbox or global state) consists of a sequence of *actions*. Currently, possible actions are the change of the global state, the sending of a message to another component,[1] or the removing of a message from the mailbox. (Note that messages are not automatically removed after reading since there may be several processes that must react on the same message.) Of course, the set of possible actions can be extended but our current set is sufficient for our case studies.

The *global state* of a component can be accessed and manipulated by all processes of the same component. Thus, it also serves as a facility for process synchronization. In general, the global state is just a tuple of data items. However, our case studies have shown that it is quite useful to partition the state into a static part with a fixed number of items and a dynamic part with an evolving number of items.[2] Therefore, we provide different actions to manipulate the static or the dynamic part of the state. The static part is changed by defining a new value for it (where changes to single items could be expressed by record updates) ("`SetState` s"). For the manipulation of the dynamic part, there are two actions: one for creating a new item ("`NewName` v ref"), which can subsequently be accessed via the newly created name ref, and one for changing the value associated to a dynamic item ("ref `:=` v"). Furthermore, there is a

[1] For the sake of simplicity, all outgoing messages are sent via the same channel. If a component wants to send messages to different other components, the messages must be tagged (to identify the receiving component) and it is the purpose of a distributor connecting the different components to forward the outgoing messages to the right receiver. Of course, one can extend our model so that a component can send messages directly to different other components but we have made the experience that our restricted model provides more modularity.

[2] Note that this distinction is not strictly necessary, since the static part can also contain dynamic data structures like lists, but useful to structure components (see the multiple counter example in Section 5).

function `get` to extract the associated value of a dynamic item in a store. Thus, the following table summarizes the current set of actions:

`Send` m	send message m
`SetState` s	set static state to s
ref `:=` v	set dynamic state object *ref* to value v
`NewName` v *ref*	create new dynamic state object *ref* with initial value v
`Deq` m	remove message m from mailbox

As described above, *processes* are activated, depending on a particular condition on the mailbox and/or global state, and perform an action followed by the creation of new processes. Thus, the behavior of each process is specified by

- a *guard* (i.e., a condition on the mailbox and/or state),
- a sequence of *actions* (to be performed when the guard is satisfied and the process is selected for execution), and
- a *process term* describing the further activities after executing the actions.

In order to structure dynamic system specifications in an appropriate manner, we allow *parameterized processes* since this supports the distinction between *local and global state*: process parameters are only accessible inside a process and, therefore, they correspond to the local state of a process, whereas the global state is visible to all processes inside a component. Changes to the local state can simply be obtained by recursive process calls with new arguments. Thus, the language of *process terms* p is very similar to process algebra [3] and defined by the following grammar:

p ::=	`Terminate`	successful termination
|	`[`a_1`,...,`a_n`]`	sequence of actions
|	`p` $t_1 \ldots t_n$	run process `p` with parameters $t_1 \ldots t_n$
|	p_1 `>>>` p_2	sequential composition
|	p_1 `<|>` p_2	parallel composition
|	p_1 `<+>` p_2	nondeterministic choice
|	p_1 `<%>` p_2	nondeterministic choice with priority
|	p_1 `<~>` p_2	parallel composition with priority

A sequence of actions is executed from left to right as one atomic operation (having a sequence of actions instead of one single action is useful to specify larger critical regions in many applications, e.g., see the dining philosophers example below). The operators "`>>>`", "`<|>`", and "`<+>`" are standard in process algebra, whereas the last two operators are not very common but useful in applications where a simple nondeterministic choice is not appropriate. The meaning of "p_1 `<%>` p_2" is: "If process p_1 can be executed, execute p_1 (and remove p_2), otherwise execute process p_2 (and remove p_1), if possible." The meaning of "p_1 `<~>` p_2" is: "Execute processes p_1 and p_2 in parallel (like "p_1 `<|>` p_2") but p_2 is executed only if p_1 cannot be executed; if p_1 terminates, then also p_2 terminates." The latter combinator is useful for idle background processes like concurrent garbage collectors.

Using this language of process terms, the behavior of a parameterized process is defined by a *process abstraction* of the following form:

$$\begin{array}{ll} \mathtt{p}\ x_1 \ldots x_n & \mid \ guard_1 \ \mathtt{=} \ actions_1 \ \mathtt{>>>} \ p_1 \\ & \vdots \\ & \mid \ guard_n \ \mathtt{=} \ actions_n \ \mathtt{>>>} \ p_n \end{array}$$

where $guard_i$ is a condition on the mailbox, the global state and the process parameters, $actions_i$ is a sequence of actions, and p_i is a process term ($i = 1, \ldots, n$). The different guards together with their right-hand sides are considered to be combined with the "<%>" operator, i.e., the first alternative with a valid guard is selected for execution.

As a first example for the use of our framework, consider the classical "dining philosophers". The global state in this example has only a static component, namely the list (or array) of `forks` where each fork has either the value `Avail` ("available") or `Used`. The entire component consists of processes `Thinking` or `Eating` that are parameterized by the number of the philosopher. Then the complete specification is as follows (`forks[`i`]` denotes the value of the i-th fork and `forks[`i`<-`v`]` denotes a new state identical to `forks` but with the value v for the i-th component):

```
Thinking i | forks[i]==Avail ∧ forks[i+1 mod n]==Avail
           = [SetState forks[i<-Used, i+1 mod n <- Used]]
             >>> Eating i

Eating i   = [SetState forks[i<-Avail, i+1 mod n <- Avail]]
             >>> Thinking i
```

Thus, if philosopher i is thinking (which corresponds to the existence of a process term "`Thinking` i") and both forks are available, then he can use both forks to turn into the `Eating` process. Note that the change of the global state, i.e., the use of both forks, can only be performed (in an atomic manner) if both forks are really available. Therefore, the classical deadlock situation is avoided without low-level synchronization (e.g., semaphores) or additional constructions (e.g., room tickets).

An example where a global state with a dynamic part becomes important will be shown later. Next we will show how this specification language can be embedded into the declarative multi-paradigm language Curry in order to obtain an executable specification language for modeling dynamic systems. Before doing so, we review the basic elements of Curry.

3 Curry

In this section we survey the elements of Curry which are necessary to understand the design and implementation of our language for specifying processes. More details about Curry's computation model and a complete description of all language features can be found in [4, 9].

Curry is a modern multi-paradigm declarative language combining in a seamless way features from functional, logic, and concurrent programming and supports programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program[3] extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Thus, a Curry program consists of the definition of functions and the data types on which the functions operate. Functions are evaluated in a lazy manner. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. The behavior of function calls with free variables depends on the evaluation annotations of functions which can be either *flexible* or *rigid*. Calls to rigid functions are suspended if a demanded argument, i.e., an argument whose value is necessary to decide the applicability of a rule, is uninstantiated ("*residuation*"). Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments to the required values in order to apply a rule ("*narrowing*").

Example 1. The following Curry program defines the data types of Boolean values and polymorphic lists (first two lines) and a function to compute the concatenation of two lists:

```
data Bool   = True | False
data List a = []   | a : List a

conc :: [a] -> [a] -> [a]
conc eval flex

conc []     ys = ys
conc (x:xs) ys = x : conc xs ys
```

The data type declarations introduce `True` and `False` as constants of type `Bool` and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell).

The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[4] Since `conc` is explicitly defined as flexible[5] (by "`eval flex`"), an equation "`conc ys [x] =:= xs`" can be solved by instantiating the first argument `ys` to the list `xs` without the last argument, i.e., for a given `xs`, the only solution to this equation satisfies that `x` is the last element of `xs`.

[3] Curry has a Haskell-like syntax [10], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

[4] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

[5] As a default, all functions except for constraints are rigid.

In general, functions are defined by (*conditional*) *rules* of the form "l | c = e" where l has the form $f\, t_1 \ldots t_n$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once, the *condition* c (which can be omitted) is a constraint, and e is a well-formed *expression* which may also contain function calls, lambda abstractions etc. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable. A *constraint* is any expression of the built-in type `Success`. Each Curry system provides at least equational constraints of the form e_1 =:= e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms (i.e., terms without defined function symbols). In contrast, e_1 == e_2 denotes an *equality test* which is successful only if both sides e_1 and e_2 are reducible to identical *ground* data terms, i.e., the test suspends in the presence of free variables.

The operational semantics of Curry, precisely described in [4, 9], is based on an optimal evaluation strategy [1] and can be considered as a conservative extension of lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Concurrent programming is supported by a concurrent conjunction operator "`&`" on constraints, i.e., a non-primitive constraint of the form "c_1 & c_2" is evaluated by solving both constraints c_1 and c_2 concurrently. Furthermore, distributed programming is supported by ports [5] which allows the sending of arbitrary data terms (also including logic variables) between different computation units possibly running on different machines connected via the Internet. The port concept has been used to integrate object-oriented features into Curry [8] and for high-level GUI (Graphical User Interface) programming in Curry [6]. Furthermore, it is relevant for the work described in this paper since the different components of a dynamic system communicate via ports (which is, however, not directly visible to the programmer).

4 Specification of Process Systems in Curry

Now we are ready to define an implementation of process-oriented specifications, as introduced in Section 2, in Curry. The implementation is guided by the motivation to enable the writing of specifications in the high-level style of Section 2. The main difference (and advantage!) is the fact that Curry is a typed language (so that we have a type checker for specifications for free) and allows definitions by pattern matching.

First, we introduce the languages of actions and process terms as data types in Curry. The following data type declaration defines the possible actions. `ObjRef` is an abstract data type denoting references to dynamic objects, and *inmsg*, *outmsg*, *static*, and *dyn* are type variables denoting the type of incoming messages, outgoing messages, the static part and the dynamic items of the global state in a concrete specifcation.

```
data Action inmsg outmsg static dyn =
           Send outmsg           -- send message
         | SetState static       -- set static state
```

```
          | Assign ObjRef dyn     -- set dynamic state object
          | NewName dyn ObjRef    -- create new dynamic object
          | Deq inmsg             -- remove message from mailbox
```

In order to support the same notation as in Section 2, we define the following function (infix operator) as a synonym for the `Assign` action:

```
ref := cont  =  Assign ref cont
```

The data type of process terms has a similar definition but with the type *proc* of concrete processes as an additional type parameter:

```
data ProcExp proc inmsg outmsg static dyn =
                  Terminate
                | Atomic [Action inmsg outmsg static dyn]
                | Proc proc
                | ParProc   (ProcExp proc inmsg outmsg static dyn)
                            (ProcExp proc inmsg outmsg static dyn)
                | SeqProc   (ProcExp proc inmsg outmsg static dyn)
                            (ProcExp proc inmsg outmsg static dyn)
                | ChProc    (ProcExp proc inmsg outmsg static dyn)
                            (ProcExp proc inmsg outmsg static dyn)
                | ChPriProc (ProcExp proc inmsg outmsg static dyn)
                            (ProcExp proc inmsg outmsg static dyn)
                | ParIdle   (ProcExp proc inmsg outmsg static dyn)
                            (ProcExp proc inmsg outmsg static dyn)
```

Again, we support the same notation as in Section 2 by the following operator definitions:

```
p1 >>> p2  =  SeqProc   p1 p2
p1 <|> p2  =  ParProc   p1 p2
p1 <+> p2  =  ChProc    p1 p2
p1 <%> p2  =  ChPriProc p1 p2
p1 <~> p2  =  ParIdle   p1 p2
```

In order to exploit the language features of Curry for the specification of dynamic systems, we consider a *system specification* as a mapping which assigns to each process, mailbox (list of incoming messages), static and dynamic state (list of dynamic objects) a process term (similarly to Haskell, a `type` definition introduces a type synonym in Curry):

```
type Specification proc inmsg outmsg static dyn =
               proc -> [inmsg] -> static -> [DynObj dyn]
                    -> ProcExp proc inmsg outmsg static dyn
```

This definition has the advantage that one can use standard function definitions by pattern matching for the specification of systems, i.e., one can define the behavior of processes in the following form:

```
spec (p x1...xn) mailbox state refs
                 | < condition on x1,...,xn, mailbox, state, refs >
```

```
= Atomic [actions] >>> process term
```

Hence, the guard is just a standard constraint on the parameters $x_1, \ldots, x_n$, `mailbox`, `state`, and `refs` so that we need no global variables or auxiliary constructs to access the current global state or mailbox (note that the access to these entities was left unspecified in Section 2).

As an example we show the complete specification of the dining philosophers of Section 2. It consists of the definition of data types for the values of forks, the philosopher processes and the definition of the specification function `phil_spec` ("`rpl` *l i v*" replaces the *i*-th element of the list *l* by *v*):

```
data ForkStatus = Avail | Used

data PhiloProc = Eating Int | Thinking Int

n = 5   -- here we have five philosophers

phil_spec (Thinking i) _ forks _
 | forks!!i == Avail && forks!!((i+1)`mod`n) == Avail
 = Atomic [SetState (rpl (rpl forks i Used) ((i+1)`mod`n) Used)]
   >>> Proc (Eating i)

phil_spec (Eating i) _ forks _ =
 Atomic [SetState (rpl (rpl forks i Avail) ((i+1)`mod`n) Avail)]
 >>> Proc (Thinking i)
```

Note that neither the mailbox nor the dynamic part of the state is used in this simple example. Initially, all philosophers are thinking. This can be expressed by a process term where five philosopher processes are combined in parallel:

```
phils = foldr1 (<|>) (map (\i->Proc (Thinking i)) [0..n-1])
```

Note that we can use standard higher-order functions like `foldr1` or `map` to create complex process terms since process terms are first-order objects in our specification language. Hence, the expression `phils` reduces to the term

```
Proc (Thinking 0) <|> ··· <|> Proc (Thinking 4)
```

A system specification is executed by providing

1. a port name for incoming messages
2. a port name for outgoing messages
3. an initial process term
4. a system specification
5. an initial (static) state[6]

This is the purpose of the main function `exec_system` so that we can execute our specification as follows:

```
exec_system "in"  "out"  phils  phil_spec
            (take n (repeat Avail))  -- all forks are available
```

[6] The dynamic part of the state is always empty at the beginning.

The main advantage of our embedding of a process-oriented language in Curry (rather than defining a complete new specification language) is the reuse of the features of Curry for the specification language, in particular:

- The type checker of Curry can be also used to type check specifications and detect inconsistencies in specifications.
- Functional programming is useful to compute values in actions, process parameters, new states etc.
- Constraint programming is useful for checking complex conditions.
- The standard abstraction facilities of Curry (e.g., higher-order functions) are useful to structure the specification of dynamic systems, in particular, we can define functions to compute process terms (compare `phils` above).

5 Examples

Due to lack of space, we can only sketch two further examples that are implemented using our framework. The first example is a challenge from the Glasgow Research Festival[7], a system of multiple counters.

The application starts be creating a single window, as shown to the right, that visualizes a counter. This counter can be manually (button "Inc") or automatically (periodically) incremented (after pressing the button "Auto"). Pressing the "Copy" button creates a new counter with its own independent state, and pressing the "Link" button creates a new view (counter window) to the same counter.

We implement this system by the specification of a *counter control system* which is responsible to control all counters and organize the communications with the different windows. If the user presses a button in a window *win*, an appropriate message (e.g., (`Inc` *win*), (`Copy` *win*)) is sent to the counter controller which must correctly react to this request. The global state of the counter controller has only a dynamic part since a new counter object is created in the state whenever the users presses the "Copy" button. The value of a counter object has the form (`Counter` *val wins mode*) where *val* is the current value of the counter, *wins* is a list of windows where this counter is displayed, and *mode* is the increment mode of the counter (`Manual` or `Automatic`). As a consequence, the individual processes of the controller are parameterized with references to counter objects and windows. For instance, there is a process `(Manual_Ctrl c w)` for each counter object `c` and window `w` where `c` is displayed. This process is responsible for processing the messages received from window `w`. Using pattern matching, there is one rule for each message in the specification. For instance, the rule for the message `Inc` is as follows:

```
cctrl (Manual_Ctrl c w) (Inc win:_) _ store | win==w
 = let Counter val windows _ = get c store in
   Atomic [c := Counter (val+1) windows Manual, Deq (Inc win)]
```

[7] `http://www.cs.chalmers.se/~magnus/GuiFest-95/`

```
    >>> Proc (Refresh_Window c) <|> Proc (Manual_Ctrl c w)
```

Thus, the guard consists of checking whether the message comes from the window for which this process is responsible. If this is the case, the value of the counter is incremented (where the increment mode is set to `Manual`), the message is removed from the mailbox, and a new process for refreshing all windows for this counter is created. The latter process sends update messages to all appropriate windows, where we apply some standard higher-order functions:

```
cctrl (Refresh_Window c) _ _ store =
   let Counter val windows _ = get c store in
   foldr (<|>) Terminate
         (map (\w->Atomic [Send (Update w val)]) windows)
```

The remaining cases are similarly defined. In particular, for each counter object `c` there is a process `(Automatic_Ctrl b c)` which is responsible for incrementing counters in automatic mode. This is done by an external clock which sends clock ticks as messages to the controller so that the `Automatic_Ctrl` processes are activated on these messages. Since there may be many of these processes, the clock tick message should not be deleted in the mailbox by any of these processes (since all of them must have the chance to react). This is the purpose of a background process `Delete_Clocks` which is simply defined as (each clock signal has a Boolean flag to distinguish successive signals):

```
cctrl Delete_Clocks (ClockSignal flag:_) _ _ =
   Atomic [Deq (ClockSignal flag)] >>> Proc Delete_Clocks
```

The complete specification of the counter controller, which is omitted due to lack of space, consists of nine rules (in addition to the three rules above, four further rules for handling the counter button messages, one rule for the `Automatic_Ctrl` process and one rule for the `Create_Window` process that creates a window together with a `Manual_Ctrl` process for it) which specify in a readable way the behavior of all processes in the controller. The initial configuration is defined by the following process term (where `c` is a free variable denoting the reference to the first counter object created by `NewName`):

```
Atomic [NewName (Counter 0 [] Manual) c] >>>
((Proc (Create_Window c) <|> Proc (Automatic_Ctrl True c))
 <~> Proc Delete_Clocks)
```

Note that it is important to create the process `Delete_Clocks` as a background process with lowest priority so that the clock signals are deleted only if no other process can be active. The remaining parts of the complete implementation, namely the counter GUIs, are also only a few lines of code thanks to the use of the Curry library for high-level GUI programming [6].

Our second example is a lift control system as visualized in Fig. 2. It consists of a number of request buttons that are inside a lift (left) or outside on the different floors (right), and a lift that can move up and down as well as open and close the doors. Instead of controlling a real lift, we simulate the lift also as a dynamic system. Thus, our implementation consists of two components in the

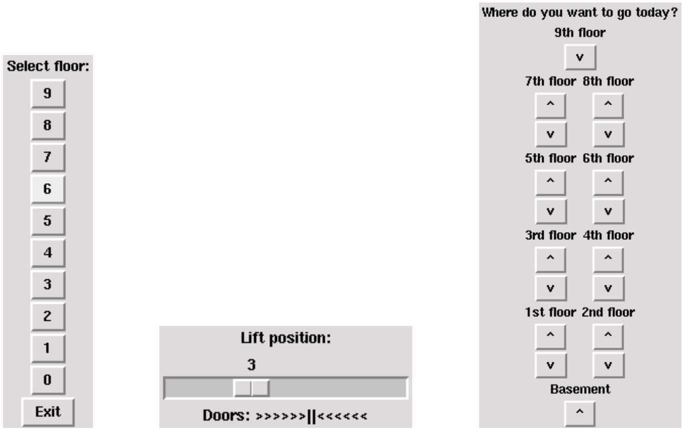


Fig. 2. A lift control system

sense of Fig. 1: a *lift controller* that accepts requests from the buttons, reacts on sensor messages from the lift (e.g., arrival at some floor), and sends appropriate control commands to the lift unit, and a *lift simulator* which simulates the lift by reacting on commands from the lift controller and sending sensor messages to the controller and the GUI (shown in the middle of Fig. 2).

The specification of the entire system can be appropriately expressed in our framework. For instance, the lift controller consists of two processes running in parallel: a process `Sorting` which is responsible to react on user requests by computing a list of floors where the lift should stop (this list is sorted according to the movement of the lift), and a second process for controlling the lift. This process can be either `Moving` or `Stopped` according to the state of the lift unit (e.g., `Moving` waits for sensor messages from the lift unit about the reached floor, and `Stopped` waits for floor requests put in by `Sorting` in the global state).

Due to lack of space, we cannot show further details from this specification, but the complete implementation is available from the authors.

6 Implementation

Our process-oriented specification language is implemented as a standard Curry library so that it can be used in any Curry program. It is freely available as a library for PAKCS (Portland Aachen Kiel Curry System) [7] and completely implemented in Curry, using the features for distributed programming [5] to implement the communication between different components of a system. The current implementation is based on an interpreter for process terms according to the operational semantics of dynamic systems (see also [2]). Although the interpreter approach is not very efficient, it is fast enough to run our examples and required only a limited implementation effort (the complete implementation consists of approximately 200 lines of Curry code, without the imported standard libraries of Curry).

7 Conclusions

We have presented a domain-specific language for process-oriented programming. Since this language is embedded in the declarative multi-paradigm language Curry, we enable process-oriented programming in Curry, which is useful for the implementation of distributed or embedded systems. On the other hand, we can reuse the programming language features of Curry for the high-level specification of dynamic systems. The specification language is based on process algebras and offers parameterized processes and a global store for the exchange of data between processes. Thus, all internal communication (synchronization) between processes is performed via the store, whereas the external communication between different dynamic systems is done by sending messages. Although our language allows high-level specifications as in other process-oriented specification languages, it is *fully executable* at the same time. Therefore, it is a useful tool to implement and test dynamic systems in a prototypical manner. We have shown the appropriateness of our framework by several case studies.

There are many proposals for process-oriented specification languages (for example, see [3]). However, as far as we know, our work is the first fully implemented approach to exploit the high-level features of both functional and logic programming for process-oriented specifications. The most similar proposal to our approach is [2] (which is not accidental since our work is inspired by many discussions with the authors of [2]). [2] contains a "generic" framework for the extension of declarative (functional, logic, functional logic) languages to include processes where there is a strict distinction between the language of processes and the underlying programming language. In particular, declarative programs are considered as the global state between transition steps of processes. Thus, the modification of declarative programs are allowed without restrictions, i.e., arbitrary program clauses can be added or deleted. This complicates the implementation of their framework. Moreover, when modifying values associated to names, the evaluation time becomes important, but this is not clearly specified in their framework. To provide an effective implementation, we have restricted all modifications to a set of well-defined data items (partitioned into a static and dynamic part of the global store). As shown by our case studies, this is sufficient for all examples discussed in [2]. Moreover, we could provide fully executable specifications of all examples, which is due to the use of the Curry libraries for distributed [5] and GUI [6] programming.

For future work we will consider more applications to study the appropriateness of our approach or necessary extensions (like real-time conditions). Furthermore, it would be interesting to consider the translation of our specification language into other existing specification or control languages in order to reuse existing verification or implementation frameworks. This would enable the use of high-level declarative programming techniques in new application fields.

Acknowledgements. The authors are grateful to Rachid Echahed and Wendelin Serwe for fruitful discussions that led to the development described in this paper.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000. Previous version in *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, 1994.
2. R. Echahed and W. Serwe. Combining Mobile Processes and Declarative Programming. In *Proc. of the 1st International Conference on Computation Logic (CL 2000)*, pp. 300–314. Springer LNAI 1861, 2000.
3. W. Fokkink. *Introduction to Process Algebra*. Springer, 2000.
4. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
5. M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pp. 376–395. Springer LNCS 1702, 1999.
6. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
7. M. Hanus, S. Antoy, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2000.
8. M. Hanus, F. Huch, and P. Niederau. An Object-Oriented Extension of the Declarative Multi-Paradigm Language Curry. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pp. 89–106. Springer LNCS 2011, 2001.
9. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.
10. J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.
11. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

HaskellMPI
Entwicklung paralleler Programme in Haskell

Michael Weber
Lehrstuhl für Informatik II
RWTH Aachen

michael.weber@i2.informatik.rwth-aachen.de

Funktionale Programmiersprachen wie ML und Haskell erlauben durch ihren hohen Abstraktionslevel sehr elegante und kurze Beschreibungen vieler Algorithmen. Durch die Syntax, die eng an mathematische Schreibweisen angelehnt ist, wird ferner die Prüfung auf Übereinstimmung einer Implementierung mit der tatsächlichen formalen Spezifikation unterstützt. Daher eignen sich funktionale Programmiersprachen insbesondere für einen *Rapid Prototyping*-Ansatz bei der Entwicklung von Algorithmen.

Aufgrund der steigenden Größe vieler Problemstellungen hat sich in den letzten Jahren die Parallelisierung von Algorithmen als wichtiger Forschungsbereich etabliert. Es ist also naheliegend, über Möglichkeiten zur Formulierung von parallelen Algorithmen in funktionalen Sprachen nachzudenken, um auch dort deren Vorteile ausnutzen zu können. Für Haskell und verwandte Programmiersprachen existieren einige Ansätze wie *Glasgow Parallel Haskell*, *Glasgow Distributed Haskell* sowie der *Eden*-Compiler, durch die Operationen zur parallelen Programmierung bereitgestellt werden. Programme, die in diesen Sprachen entwickelt werden, erfordern jedoch ein spezielles Laufzeitsystem, das vom jeweiligen Compiler eingebunden wird. Weiterhin sind diese Programme aufgrund syntaktischer Unterschiede der Sprachen nicht zwischen verschiedenen Compiler-Systemen austauschbar.

In diesem Vortrag wird mit HaskellMPI eine Bibliothek vorgestellt, mit der man in Standard-Haskell die Möglichkeit erhält, parallele Programme zu schreiben. Diese sind nicht abhängig von speziellen Haskell-Implementierungen, so daß zur Unterstützung eines weiteren Compilers lediglich die HaskellMPI-Bibliothek portiert werden muß. Ferner werden einige Beispiele für die Anwendung der Bibliothek gegeben.

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

Mythen und Fakten über Basisblockgraphen

Markus Mohnen
RWTH Aachen

mohnen@informatik.rwth-aachen.de

Zusammenfassung Datenflußanalyse auf der Basis von Basisblockgraphen ist seit Jahren die vorherrschende Technik zur sicheren Approximation von Programmeigenschaften. In der neueren Forschung wurden allerdings Zweifel bezüglich der Adäquatheit dieser Wahl geäußert. Dabei wurden die aus konzeptioneller Sicht einfacheren Instruktionsgraphen als Alternative diskutiert. Offen blieb dabei allerdings die Frage nach der in der Praxis auftretenden Effizienz der Ansätze. Diese Frage kann auf konzeptioneller Ebene nicht abschließend beantwortet werden, da die Effizienz vom zu erwartenden „Durchschnittsfall" abhängt.
In diesem Vortrag wird eine ausführliche empirische Bewertung der beiden Ansätze vorgestellt. Grundlage dabei ist eine mit 14.744 Klassen und 98.101 Methoden sehr umfangreiche Sammlung von Java Klassendateien. Diese große Grundlage ermöglicht es uns den zu erwartenden „Durchschnittsfall" besser vorherzusagen.

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

Distributing and Managing Mobile Code (Abstract)

Michael Franz

University of California, Irvine

Using mobile code is fraught with risks. If an adversary deceives us into executing a malicious program, this may have catastrophic consequences and may lead to a loss of confidentiality, loss of information integrity, loss of the information itself, or a combination of these outcomes. Unfortunately, additional provisions for security frequently cause a loss of efficiency, often to the extent of making an otherwise virtuous security scheme unusable for all but "toy" programs. Moreover, current mobile-code distribution models are surprisingly primitive; for example, they assume that all constituent parts that make up a mobile program are downloaded to a single location, then verified, linked, possibly dynamically compiled, and finally executed at that same location.

Our research in the past 6 years has focused on making mobile code *practical*, to the extent that it can completely displace native code for the whole spectrum of applications from PDAs to desktop applications to scientific computing on supercomputers.

Our contributions include work on dynamic compilation, which raises the performance of such programs, often to the extent of outperforming statically compiled code.

A second major contribution is a class of representations for target-machine independent mobile programs that can provably encode only legal programs. Hence, there is no way an adversary can substitute a malicious program that can corrupt its host computer system: Every well-formed mobile program that is expressible in our encoding is guaranteed to map back to a source program that is deemed legal in the original source context, and mobile programs that are not well-formed can be rejected trivially. Further, our encoding not only guarantees referential integrity and type-safety within a single distribution module, but it also enforces these properties across compilation-unit boundaries. As a side-effect, our encoding is exceptionally dense: it outperforms the next best compression scheme for Java by almost 50%.

A third focus of research is on managing the mobile-code pipeline from code producer to code consumer: capturing all meaningful modes of mobile-code deployment in a model, and then creating a code management architecture that makes this model enforceable by mechanical means.

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

An Implementation of Narrowing Strategies*

Sergio Antoy[1] Michael Hanus[2] Bart Massey[1] Frank Steiner[2]

[1] Department of Computer Science, Portland State University,
P.O. Box 751, Portland, OR 97207, U.S.A.
{antoy,bart}@cs.pdx.edu
[2] Institut für Informatik, Christian-Albrechts-Universität Kiel,
Olshausenstr. 40, D-24098 Kiel, Germany
{mh,fst}@informatik.uni-kiel.de

Abstract. This paper describes an implementation of narrowing, an essential component of implementations of modern functional logic languages. These implementations rely on narrowing, in particular on some optimal narrowing strategies, to execute functional logic programs. We translate functional logic programs into imperative (Java) programs without an intermediate abstract machine. A central idea of our approach is the explicit representation and processing of narrowing computations as data objects. This enables the implementation of operationally complete strategies (i.e., without backtracking) or techniques for search control (e.g., encapsulated search). Thanks to the use of an intermediate and portable representation of programs, our implementation is general enough to be used as a common back end for a wide variety of functional logic languages.

1 Introduction

This paper describes an implementation of narrowing for overlapping inductively sequential rewrite systems [5]. Narrowing is the essential computational engine of functional logic languages (see [13] for a survey on such languages and their implementations). An implementation of narrowing translates a program consisting of rewrite rules into executable code. This executable code currently falls into two categories: Prolog predicates (e.g., [4, 11, 14, 25]) or instructions for an abstract machine (e.g., [10, 18, 24, 27]). Although these approaches are relatively simple, in both cases, several layers of interpretation separate the functional logic program from the hardware intended to execute it. Obviously, this situation does not lead to efficient execution.

In this paper we investigate a different approach. We translate a functional logic program into an imperative program. Our target language is Java, but we

* This research has been partially supported by the DAAD/NSF under grant INT-9981317 and the German Research Council (DFG) under grant Ha 2457/1-2. This paper is an abridgement of a paper to appear in the proceedings of the *Third International Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, and is copyright 2001 by the Association for Computing Machinery. Extracts of that paper are reproduced for this purpose by permission of the ACM.

make limited use of specific object-oriented features, such as inheritance and dynamic polymorphism. Replacing Java with a lower-level target language, such as C or machine code, would be a simple task.

In Section 2 we briefly introduce the aspects of functional logic programming relevant to our discussion. In Section 3 we describe the elements and the characteristics of our implementation of narrowing. In Section 4 we describe aspects of our compilation process, as well as execution issues such as input, output and tracing/debugging that may greatly affect the usability of a system. In Section 5 we summarize current efforts toward the implementation of functional logic languages, particularly w.r.t. implementations of narrowing and how they compare to our work. Section 6 offers some conclusions.

2 Functional Logic Programs

Functional logic languages combine the operational principles of two of the most important declarative programming paradigms, namely functional and logic programming (see [13] for a survey). Efficient demand-driven functional computations are amalgamated with the flexible use of logical variables, providing for function inversion and search for solutions. Functional logic languages with a sound and complete operational semantics are usually based on narrowing (originally introduced in automated theorem proving [29]) which combines reduction (from the functional part) and variable instantiation (from the logic part). A *narrowing step* instantiates variables of an expression and applies a reduction step to a redex of the instantiated expression. The instantiation of variables is usually computed by unifying a subterm of the entire expression with the left-hand side of some program equation.

Example 1. Consider the following rules defining the $\leq$ predicate **leq** on natural numbers which are represented by terms built from **zero** and **succ**:

```
leq(zero,Y)           = true
leq(succ(X),zero)     = false
leq(succ(X),succ(Y))  = leq(X,Y)
```

The expression **leq(succ(M),Y)** can be evaluated (i.e., reduced to a value) by instantiating **Y** to **succ(N)** to apply the third equation, followed by the instantiation of **M** to **zero** to apply the first equation:

$$\texttt{leq(succ(M),Y)} \leadsto_{\{\texttt{Y}\mapsto\texttt{succ(N)}\}} \texttt{leq(M,N)} \leadsto_{\{\texttt{M}\mapsto\texttt{zero}\}} \texttt{true}$$

Narrowing provides completeness in the sense of logic programming (computation of all answers, i.e., substitutions leading to successful evaluations) as well as functional programming (computation of values). Since simple narrowing can have a huge search space, a lot of effort has been made to develop sophisticated narrowing strategies without losing completeness (see [13]). *Needed narrowing* [7] is based on the idea of evaluating only subterms which are *needed* in order to compute a result. For instance, in a term like **leq**(t_1,t_2), it is always necessary to evaluate t_1 (to some variable or constructor-rooted term) since all three rules in Example 1 have a non-variable first argument. On the other hand, the evaluation of t_2 is only needed if t_1 is of the form **succ(** t **)**. Thus, if t_1 is a free variable, needed narrowing instantiates it to a constructor term, here **zero** or **succ(V)**. Depending on this instantiation, either the first equation is applied or the second argument t_2 is evaluated. Needed narrowing is currently the best narrowing strategy for first-order (inductively sequential) functional logic programs [3] due to its optimality properties w.r.t. the length of derivations and the independence of computed solutions, and due to the possibility of efficiently implementing needed narrowing by pattern matching and unification [7]. Moreover, it has been extended in various directions, e.g., higher-order functions and λ-terms as data structures [17], overlapping rules [5], and concurrent computations [15].

Needed narrowing is complete, in the sense that for each solution to a goal there exists a narrowing derivation computing a more general solution. However, most of the existing implementations of narrowing lack this property since they are based on Prolog-style backtracking. Since backtracking is not fair in exploring all derivation paths, some solutions might not be found in the presence of infinite derivations, i.e., these implementations are incomplete from an operational point of view. An important property of our implementation is its operational completeness, i.e., all computable answers are eventually computed by our implementation.

3 Implementation of Needed Narrowing

In this section we describe the main ideas of our implementation of narrowing. We implement a strategy, referred to as *INS* [5], proven sound and complete for the class of the overlapping inductively sequential rewrite systems. In these systems, the left-hand sides of the rewrite rules defining an operation can be organized in definitional trees. However, an operation may have distinct rewrite rules with the same left-hand side (modulo renaming of variables): operation **coin** (Section 3.8), is one example. To ease the understanding of our work, we first describe the implementation of rewrite computations in inductively sequential rewrite systems. We then describe the extensions that lead to narrowing in overlapping inductively sequential rewrite systems.

3.1 Overview

The overall goals of our implementation are speed of execution and operational completeness. The following principles guide our implementation and are instrumental in achieving the goal.

1. A reduction step replaces a redex of a term with its reduct. A term is represented as a tree-like data structure. The execution of a reduction updates only the portion of this data structure affected by the replacement. Thus, the cost of a reduction is independent of its context. We call this principle *in-place* replacement.

2. Only somewhat needed steps are executed. We use the qualifier "somewhat" because different notions of *need* have been proposed for different classes of rewrite systems. We execute a particular kind of steps that for reductions in orthogonal systems is known as *root-needed* [28]. Thus, reductions that are a priori useless are never performed. We call this principle *useful step.*
3. *Don't know* non-deterministic reductions are executed in parallel. Both narrowing computations (in most rewrite systems) and reductions (in interesting rewrite systems) are non-deterministic. Without some form of parallel execution, operational completeness would be lost. We call this principle *operational completeness.*

In inductively sequential rewrite systems, and when computations are restricted to rewriting, it is relatively easy to faithfully implement all the above principles. In fact, our implementation does it. However, our environment is considerably richer. We execute *narrowing* computations in *overlapping* inductively sequential rewrite systems. In this situation, two complications arise. The non-determinism of narrowing and/or of overlapping rules imply that a redex may have several replacements. In these situations, there cannot be a single in-place replacement. Furthermore, the steps that we compute in *overlapping* inductively sequential rewrite systems are needed, but only modulo non-deterministic choices [5]. Hence, some step may not be needed in the strict sense of [7, 22], but we may not be able to know by feasible means which steps.

The architecture of our implementation is characterized by *terms* and *computations.* Both terms and computations are organized into tree-like linked (dynamic) structures. A *term* consists of a *root symbol* applied to zero or more *arguments* which are themselves terms. A *computation* consists of a stack of *terms* that identify reduction steps. All the terms in the stack, with the possible exception of the top, are not yet redexes, but will eventually become redexes, and be reduced, before the computation is complete. In terms, links go from a parent to its children, whereas in computations links go from children to their parent.

A graphical representation of these objects is shown in Figure 1. In this figure, the steps to the left represent the terms in the stack of the computation. $Step_0$ is the bottom of the stack: it cannot be executed before $Step_1$ is executed. Likewise $Step_1$ cannot be executed before $Step_2$ is executed.

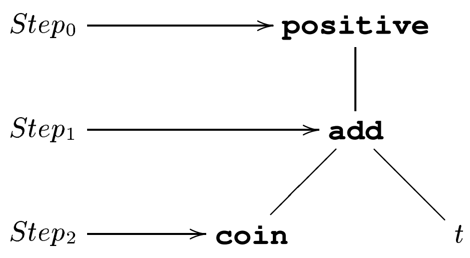


Fig. 1. Snapshot of a computation of term **`positive(add(coin,`***t***`))`**

To ease understanding, we begin with an account of our implementation of rewriting computations in inductively sequential rewrite systems. Although non-trivial, this implementation is simple enough to inspire confidence in both its correctness and efficiency. Then, we generalize the discussion to larger classes of rewrite systems and finally to narrowing computations and argue why both correctness and efficiency of this initial implementation are preserved by these extensions.

3.2 Symbol representation

Symbols are used to represent terms. A *symbol* is an object that contains two pieces of information: a *name* and a *kind*. Since there is no good reason to have more than one instance of a given symbol in a program, each distinct symbol is implemented as an immutable singleton object. The *name* is a string. The *kind* is a tag that classifies a symbol. For now, the tag is either "defined operation" or "data constructor". Additional tags will be defined later to compute with larger classes of rewrite systems. The tag of a symbol is used to dispatch computations that depend on the classification of a symbol. Of course, we could dispatch these computations by dynamic polymorphism, i.e., by defining an abstract method overridden by subclasses. Often, these methods would consist of a few statements that use the environment of the caller. A tag avoids both a proliferation of small methods and the inefficiency of passing around the environment. Furthermore, this architecture supports implementations in objectless target languages as well.

Nevertheless, in our Java architecture, class *symbol* has subclasses such as *operation* and *constructor*. In particular, there is one subclass of *operation* for each defined operation f of a functional logic program. This class, according to our second principle, contains the code for the execution of a useful step of any term rooted by f. Operations are defined by rewrite rules. We use the following rules in the examples to come.

```
add (zero, Y)     = Y
add (succ (X), Y) = succ (add (X, Y))

positive (zero)     = false
positive (succ (_)) = true
```

3.3 Term representation

Terms of user-defined type contain two pieces of information: the *root* of the *term*, which is a *symbol*, and the *arguments* of the *root*, which are *terms* themselves. Terms of builtin types contain specialized information, e.g., terms of the builtin type *int* contain an *int*. This situation suggests defining a common base class and a specialization of this class for each appropriate type of term. However, this is in conflict with the fact that according to the first principle of our implementation, a *term* is a mutable object. In Java, the class of an object cannot change during execution.

Therefore, we implement a *term* as a bridge pattern. A term delegates its functionality to a representation. Different types, such as user-defined types, builtin types, and variables are represented differently. All the representations provide a common functionality. The representation of a term object can change at run-time and thus provide mutability of both value and behavior as required by the implementation.

3.4 Computation representation

A *computation* is an object abstracting the necessity to execute a sequence of specific reduction steps in a term. Class *computation* contains two pieces of information:

1. A *stack of terms* to be contracted (reduced at the root). The terms in the stack are not redexes except, possibly, the top term. Each term in the stack is a subterm of the term below it, and must be reduced to a constructor-rooted term in order to reduce the term below it. Therefore, the elements of the stack in a computation may be regarded as steps as well. The underpinning theoretical justification of this stack of steps is in the proof of Th. 24 of the extended version of [5]. We ensure that every term in the stack eventually will be contracted. To achieve this aim, if a complete strategy cannot execute a step in an operation-rooted term, it reduces the term to the special value *failure*.
2. A set of *bookkeeping information*. For example, this information includes the number of steps executed by the computation and the elapsed time. An interesting bookkeeping datum is the state of a computation. Computations being executed are in a *ready* state. A computation's state becomes *exhausted* after the computation has been executed and it has been determined that no more steps will be executed at the root of the bottom-most term of the stack. Before becoming exhausted a computation state may be either *result* or *failure*. Later, we will extend our model of computation with residuation. With the introduction of residuation, a new state of a computation, *flounder*, is introduced as well.

Loosely speaking, an initial computation is created for an initial top-level expression to evaluate. This expression is the top and only term of the stack of this computation. If the top term t is not a redex, a subterm of t needed to contract t is placed on the stack and so on until a redex is found. A redex on top of the stack is replaced by its reduct. If the reduct is constructor-rooted, the stack is popped (its top element is discarded).

3.5 Search space representation

The search space is a queue of computations which are repeatedly selected for processing. The machinery of a queue and fair selection is not necessary for rewriting in inductively sequential rewrite systems. For these systems, computations are strictly sequential and consequently a single (possibly implicit) stack of steps would suffice. However, the architecture that we describe not only accommodates the extensions from rewriting to narrowing and/or from inductively sequential rewrite systems to the larger classes that are coming later, but it allows us to compute more efficiently.

A computation serves two purposes: (1) finding maximal operation-rooted subterms t of the top-level term to evaluate and (2) reducing each t to head normal form. The pseudo-code of Figure 2 sketches part (2), which is the most challenging. Some optimizations would be possible, but we avoid them for the sake of clarity.

Since inductively sequential rewrite systems are confluent, replacing in-place a subterm u of a term t with u's reduct does not prevent reaching t's normal form. When a term has a result this result is found, since repeated contractions of needed redexes are normalizing.

```
while the queue is not empty
| select a ready computation k from the queue
| let t be the term at the top of k's stack
| switch on the root of t
| | case t is operation-rooted
| | | switch on the reducibility of t
| | | | case t is a redex
| | | | | replace t with its reduct
| | | | | put k back into the queue
| | | | case t is not a redex
| | | | | switch on s, a maximal needed subterm of t
| | | | | | case s exists
| | | | | | | push s on k's stack
| | | | | | | put k back into the queue
| | | | | | case s does not exist
| | | | | | | stop the computation, no result exists
| | | | | endswitch
| | | endswitch
| | case t is constructor-rooted
| | | pop k's stack
| | | if k's stack is not empty
| | | | put k back into the queue
| endswitch
endwhile
```

Fig. 2. Procedure to evaluate a term to a head normal form

3.6 Sentinel

The first extension to the previous model is the introduction of a "sentinel" at the root of the top-level expression being evaluated. For this, we introduce a distinguished symbol called *sentinel* that takes exactly one argument of any kind. If t is the term to evaluate, our implementation evaluates $sentinel(t)$ instead. Thus, this is the actual term of the initial computation. Symbol *sentinel* has characteristics of both an operation and a constructor. Similar to an operation, the stack of the initial computation contains $sentinel(t)$, but similar to a constructor, $sentinel(t)$ cannot be contracted for any t. Having a sentinel has several advantages. The strategy works with the sentinel by means of implicit rewrite rules that always look for an internal needed redex and never contract the *sentinel*-rooted term itself. Also, using a sentinel saves frequent tests similar to using a sentinel in many classic algorithms, e.g., sorting.

3.7 Failure

The second extension to the previous model is concerned with the possibility of a "failure" of a computation. A failure occurs when a term has no constructor normal form. The computation detects a failure when the strategy, which is complete, finds no useful steps (redexes) in an operation-rooted term.

The pseudo-code presented earlier simply terminates the computation when it detects a failure. For the extensions discussed later it is more convenient to explicitly represent failures in a term. This allows us, e.g., to clean up computations that cannot be completed and to avoid duplicating certain computations. To this purpose we introduce a new symbol called *failure*. The *failure* symbol is treated as a constant constructor.

Suppose that u is an operation-rooted term. If the strategy finds no step in u, it evaluates u to *failure*. A *failure* symbol is treated as a constructor during the pattern matching process. Implicit rewrite rules for each defined operation rewrite any term t to *failure* when a *failure* occurs at a needed position of t. For example, we perform the following reduction:

add (failure, v**)** $\to$ **failure**

With these implicit rewrite rules, an inner occurrence of *failure* in a term propagates up to the sentinel, which can thus report that a computation has no result. The explicit representation of failing computations is also important in performing non-deterministic computations.

3.8 Non-determinism

The third extension to the previous model is concerned with non-determinism. In our work, non-determinism is expressed by rewrite rules with identical left-hand sides, but distinct right-hand sides. A textbook example of a non-deterministic defined operation is:

```
coin = zero
coin = succ (zero)
```

This operation differs from the previous ones in that a given term, say $s =$ **coin**, has two distinct reducts.

The most immediate problem posed by non-deterministic operations is that if s occurs in some term t and we replace in-place s with one of its replacements, we may lose a result that could be obtained with another replacement. If a term such as s becomes the top of the stack of a computation k, we change the state of k to *exhausted* and we start two or more new computations. Each new computation, say k', begins with a stack containing a single term obtained by one of the several possible reductions of s.

The procedure described above can be optimized in many ways. We mention only the most important one that we have implemented — the sharing of subterms disjoint from s. We show this optimization in an example. Suppose that the top-level term being evaluated is:

`add (coin,` t**`)`**

The non-determinism of **`coin`** gives rise to the computation of the following two terms:

`add (zero,` t**`)`**
`add (succ (zero),` t**`)`**

These terms are evaluated concurrently and independently. However, term t in the above display is shared rather than duplicated. Sharing improves the efficiency of computations since only one term, rather than several equal copies, is constructed and possibly evaluated. In some situations, a shared term may occur in the stacks of two independent computations and be concurrently evaluated by each computation. This approach avoids a common problem of backtracking-based implementations of functional logic languages, in which t will be evaluated twice if it is needed during the evaluation of both **`add`** terms shown above.

3.9 Rewrite rules

The final relevant portion of our architecture is the implementation of rewrite rules. All the rules of an ordinary defined operation f are translated into a single Java method. This method implicitly uses a definitional tree of f to compare constructor symbols in inductive positions of the tree with corresponding occurrences in an f-rooted term t to reduce. Let k_t be a computation in the queue, *ready* the state of k_t, and t the term on the top of k_t's stack. The following case breakdown defines the code that needs to be generated.

1. If t is a redex with a single reduct, then t is replaced in-place by its reduct.
2. If t is a redex with several reducts, then a new computation is started for each reduct. The state of k_t is changed to *exhausted*.
3. If in a needed position of t there is *failure*, then t is considered a redex as well and it is replaced in-place by *failure*.
4. If in a needed position of t there is an operation-rooted ordinary term s, then s is pushed on the stack of k_t.
5. The last case to consider is when operation f is incompletely defined and no needed subterm is found in t. In this case, t is replaced in-place by *failure*.

3.10 Narrowing

At this point we are ready to discuss the extension of our implementation to narrowing. A narrowing step instantiates variables in a way very similar to a non-deterministic reduction step. For example, suppose that *allnat* is an operation defined by the rules:

```
allnat = zero
allnat = succ (allnat)
```

Narrowing term **add(X,** t**)**, where **X** is an uninstantiated variable and t is any term, is not much different from reducing **add(allnat,** t**)**.

There are two key differences in the handling of variables w.r.t. non-deterministic reductions: (1) we must keep track of variable bindings to construct the *computed answer* at the end of a computation, and (2) if a given variable occurs repeatedly in a term being evaluated, the replacement of a variable with its binding must replace all the occurrences. We solve point (1) by storing the binding of a variable in a computation. Point (2) is simply bookkeeping. We represent substitutions "incrementally." A computation computes both a value (for the functional part) and an answer (for the logic part). The answer is a substitution. In most cases, a narrowing step produces several distinct bindings for a variable. Each of these bindings increments a previously computed substitution. For example, suppose that the expression to narrow is:

add (X, Y) = t

for some term t. Some computation may initially bind **X** to **zero**. Later on, a narrowing step may bind **Y** independently to both **zero** and **succ(**Y_1**)**. These bindings will "add" to the previous one. The previous binding is shared, which saves both memory and execution time.

3.11 Parallelism

Our implementation includes a form of parallelism known as *parallel-and.* And-parallel steps do not affect the soundness or completeness of the strategy, *INS*, underlying our implementation, but in some cases they may significantly reduce the size of the narrowing space of a computation — possibly from infinite to finite. The *parallel-and* operation is handled explicitly by our implementation. If a computation k leads to the evaluation of t **&** u, where t and u are terms and "**&**" denotes the parallel-and operation, then steps of both t and u are scheduled. This requires to change the stack of a computation into a tree-like structure. The set of leaves of this tree-like structure replaces the top of the stack previously discussed.

As soon as one of these parallel steps has to be removed from the tree, which means that its term argument has been reduced to a constructor term c (including *failure*), the parent of the step is reconsidered. Depending on c's value, either the parent term is reduced (to a *failure* if c = *failure*) and the other parallel steps are removed, or (if c = *success*) the computation of the other parallel steps continues normally.

3.12 Residuation

Residuation is a computational mechanism that delays the evaluation of a term containing an uninstantiated variable in a needed position [1]. Similar to narrowing, it supports the integration of functional programming with logic programming by allowing uninstantiated variables in functional expressions. However, in contrast to narrowing it is incomplete, i.e., unable to find all the solutions of some problems. Residuation is useful for dealing with built-in types such as numbers [9]. Residuation is meaningful only when a computation has several steps executing in parallel. If a computation has only one step executing, and this step residuates, the computation cannot be completed and it is said to *flounder*.

Operations that residuate are called *rigid*, whereas operations that narrow are called *flexible*. A formal model for the execution of programs defining both rigid and flexible operations is described in [15]. Our implementation already has the necessary infrastructure to accommodate this model. When a step s residuates on some variable V, we store (a reference to) s in V, mark s as *residuating* and continue the execution of the other steps. When V is bound, we remove the *residuating* mark from s so that s can be executed as any other step. If all the steps of a computation are *residuating*, the computation *flounders*.

4 The Compilation Process

The main motivation of this new implementation of narrowing is to provide a generic back end that can be used by functional logic languages based on a lazy evaluation strategy. Current work [6] shows that any narrowing computation in a left-linear constructor-based conditional rewrite system can be simulated, with little or no loss of efficiency, in an overlapping inductively sequential rewrite system, hence by our implementation. Therefore, our implementation can be used by languages such as Curry [20], Escher [23] and Toy [26].

To support this idea, our implementation works independently of any concrete source language. The source programs of our implementation are functional logic programs where all functions are defined at the top level (i.e., no local declarations) and the pattern-matching strategy is explicit. This language, called FlatCurry, has been developed as an intermediate language for the Curry2Prolog compiler [8] in the Curry development system PAKCS [16] and is used for various other purposes, e.g., meta-programming and partial evaluation [2]. Basically, a FlatCurry program is (apart from data type and operator declarations) a list of function declarations where each function f is defined by a single rule of the form $f(x_1, \ldots, x_n)$ `=` e, i.e., the left-hand side consists of pairwise different variable arguments and the right-hand side is an expression containing case expressions for pattern matching.

For instance, the function **`leq`** of Example 1 is represented in FlatCurry as follows (*fcase* denotes a case expression that is evaluated by narrowing):

$$
\begin{array}{l}
\texttt{leq(X,Y)} = \mathit{fcase}\ \texttt{X}\ \mathit{of}\ \{\ \texttt{zero} \rightarrow \texttt{true};\\
\qquad\qquad\qquad\qquad\qquad \texttt{succ(M)} \rightarrow \mathit{fcase}\ \texttt{Y}\ \mathit{of}\ \{\texttt{zero} \rightarrow \texttt{false};\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{succ(N)} \rightarrow \texttt{leq(M,N)}\}\}
\end{array}
$$

A detailed description of FlatCurry including constructs for encoding features like non-deterministic choices (see Section 3.8), residuation (see Section 3.12), higher-order functions or conditional rules can be found on the Curry webpage at `http://www.informatik.uni-kiel.de/~curry/flat/`. Any inductively sequential program can be translated into FlatCurry rules whose right-hand side consists of only constructor/function applications and case expressions [17].

Although FlatCurry was originally designed as an intermediate language to compile and manipulate Curry programs, it should be clear that it can also be used for various other declarative languages (e.g., Haskell-like lazy languages with strict left-to-right pattern matching can be compiled by generating appropriate case expressions). To better accommodate a variety of source languages, our back end accepts a syntactic representation of FlatCurry programs in XML format so that other functional logic languages can be compiled into this implementation-independent format. Some examples together with the DTD for the XML FlatCurry representation are available at `http://www.informatik.uni-kiel.de/~curry/flat/`.

Our compiler, which is fully implemented in Curry, reads an XML representation and compiles it into a Java program following the ideas described in Section 3. Recall that every function is represented by a subclass of *operation*. For each function, we define a method *expand* which will expand a function call according to its rules and depending on its arguments (Sections 3.9, 3.10).

To show the simplicity of our compiled code, we provide an excerpt of the *expand* method for **`leq`** in Figure 3 which is generated from the case expression given above. According to Section 3.9, we must decide whether **`leq(`**t_1, t_2**`)`** is a redex. This expression is a redex if t_1 is a variable (we must narrow) or **`zero`** (we apply the first rule). If t_1 equals **`succ(..)`**, we must do the same check for the second argument. If t_1 fails, so does **`leq`**. If t_1 is a function call, we must evaluate it first. For the sake of simplicity, we show pseudo-code, which reflects the basic structure and is very similar to the real Java code.

To use our back end for a functional logic language, it is only necessary to compile programs from this language to a XML representation according to the FlatCurry DTD. For instance, our compiler can be used as a back end for Curry since Curry programs can be translated into this XML representation with PAKCS [16]. Again, it is worth emphasizing that FlatCurry can encode more than just Curry programs or needed narrowing, because the evaluation strategy is compiled into the case expressions. For instance, FlatCurry is a superset of TFL, which is used as an intermediate representation for a Toy-like language based on the CRWL paradigm (Constructor-based conditional ReWriting Logic) [21].

The computation engine is designed to work with the *read-eval-print* loop typical of many functional, logic and functional logic interpreters. In our Java implementation, the computation engine and the read-eval-print loop are threads that interact with each other in a producer/consumer pattern. When a computed expression (value plus answer) becomes available, the computation engine notifies the read-eval-print loop while preserving the state of the narrowing space.

```
expand (Computation comp) {
    term = comp.getTerm();             //  get the term from top of the stack
    X = term.getArg(0);                // get first argument
    Y = term.getArg(1);                // get second argument
    switch on kind of X                // case X of ...
    case variable:                     // do narrowing: bind to patterns
        X.bindTo(zero);
        spawn new computation for leq(zero,Y);
        X.bindTo(succ(M));
        spawn new computation for leq(succ(M),Y);
        comp.setExhausted();              // this computation is exhausted
    case constructor:                     // argument is constructor-rooted,
        switch on kind of constructor     // thus do pattern matching
        case zero:                        // apply first rule:
            term.update(true);            // replace term with true
        case succ:                        // case X of succ(M) → case Y of...
           recursive case for switching on Y
    case failure:                         // the needed subterm has failed,
        term.update(failure)              // thus leq fails, too
    case operation:                       // X is a function call, thus
        comp.pushOnStack(X);              // evaluate this call first
}
```

Fig. 3. Simplified pseudo-code for the `expand` method of **leq**

The read-eval-print loop presents the results to the user and waits. The user may request further results or terminate the computation. If the user requests a new result, the read-eval-print loop notifies the computation engine to further search the narrowing space. Otherwise, the narrowing space is discarded.

Currently we provide a naive trace facility that is useful to debug both user code and our own implementation. Since the computations originating from a goal are truly concurrent, as is necessary to ensure operational completeness, and since some terms are shared between computations, the trace is not always easy to read. Computations are identified by a unique *id*. We envision a tool, conceptually and structurally well separated from the computation engine, that collects the interleaved traces of all computations, separates them, and presents each trace in a different window for each computation. This tool may have a graphical user interface to select which computations to see and/or interact with.

5 Related work

In this section we discuss and compare other approaches to functional logic language implementation (see [13] for a survey). Our approach provides an operationally complete and efficient architecture for implementing narrowing which can potentially accommodate sophisticated concepts, e.g., the combination of narrowing and residuation, encapsulated search or committed choice. As some recent narrowing-based implementations of functional logic languages show, most implementations that include these concepts lack completeness or are inefficient.

One common approach to implement functional logic languages is the transformation of source functional logic programs into Prolog programs. This approach is favored for its simplicity since Prolog has most of the features of functional logic languages: logical variables, unification, and non-determinism implemented by backtracking. However, the challenge in such an implementation is the implementation of a sophisticated evaluation strategy that exploits the presence of functions in the source programs. Different implementations of this kind are compared and evaluated in [14] where it is demonstrated that needed narrowing is efficiently implemented in a (strict) language such as Prolog and that this implementation is superior to other narrowing strategies. Therefore, most of the newer proposals to implement functional logic languages in Prolog are based on needed narrowing [4, 8, 14, 25]. In contrast to our implementation of narrowing, all of these efforts are operationally incomplete (i.e., existing solutions might not be found due to infinite derivation paths) since they are based on Prolog's depth-first search mechanism. The same drawback also occurs in implementations of functional logic languages based on abstract machines (e.g., [10, 24, 27, 21]) since these abstract machines use backtracking to implement non-determinism.

An exception is the Curry2Java compiler [18] which is based on an abstract machine implementation in Java but uses independent threads to implement non-deterministic choices. If these threads are fairly evaluated (which can be ensured by specific instructions), infinite derivations in one branch do not prevent finding solutions in other branches. Our approach is more flexible since it does not depend on threads, but it can control to any degree of granularity the scheduling of steps in distinct computations. This eases the implementation of problem-specific search strategies at the top level, whereas Curry2Java is restricted to encapsulated search [19].

Our implementation is the subject of active investigation in several directions. Thus, we are not specifically concerned with its efficiency at this time. Rather, we are studying architectures that easily integrate concepts and ideas that have been proposed for functional logic programming. Efficiency is an important issue, though, and we expect that it will be a strong point of our implementation due to the direct translation into an imperative language without the additional control layers of an abstract machine. While we have attempted to select an efficient architecture, we have not paid much attention to detailed optimization of our implementation, and we do not expect top speed as long as we compile to Java. We performed only a limited number of benchmarks to get a feel for where we stand.

For the functional evaluation, we evaluated the naive reverse of a list of 1200 elements (400 only for comparing Curry2Java). To benchmark non-determinism we evaluated **`add x y =:= peano300`**, where **`peano300`** denotes the term encoding 300 in unary notation and the infix operator **`=:=`** denotes the strict equality with unification. This goal is solved by creating 301 parallel computations by narrowing on the **`add`** operation.

Table 1. Execution times for simple benchmarks on several FLP engines

	Ours	C2J	MCC	PAKCS	Jinni
$\mathbf{rev}_{400}$	0.69	2.6			
$\mathbf{rev}_{1200}$	5.5	N/A	0.69	0.68	45.9
$\mathbf{add}_{300}$	2.1	16.2	0.12	0.09	2.5

The two fastest available implementations of needed narrowing, to the best of our knowledge, are the Curry2Prolog compiler of the PAKCS system and the *Münster Curry Compiler (MCC)* [27]. The Curry2Java back end (C2J), included in the PAKCS system, is not as fast, but is the fastest available correct and complete implementation of needed narrowing. We have also compared our approach to a Java-based implementation of Prolog: Jinni [30] is the fastest engine in the naive reverse benchmark among the Java-based Prolog implementations compared in [12]. Table 1 shows execution times, in seconds, for simple benchmarks on a PIII-900 MHz Linux machine. These results show that our engine is currently the fastest *complete* implementation of narrowing.

In all likelihood, its speed is partially due to the elimination of the overhead paid by Curry2Java for computing with an abstract machine. In comparison with Jinni, we perform better in the $\mathbf{rev}_{1200}$ benchmark, where the number of reduction steps is more or less the same for needed narrowing and SLD-resolution. For the **add** benchmark, we evaluate the goal **add(X,Y,peano300)** in Jinni. Due to the rules for strict equality with unification, even an optimized implementation of needed narrowing will perform at least twice as many reduction steps for **add x y =:= peano300** as a SLD-resolution of **add(X,Y,peano300)**. However, we are still faster than Jinni in this benchmark, too. Curry2Prolog and MCC are faster than our approach by a factor 8 for **rev** and by factor 20 for **add**. This is to be expected. Backtracking-based implementations are simpler and faster because they sacrifice completeness. Additionally, Curry2Prolog is executed by the highly optimized SICStus Prolog compiler, and the abstract machine of MCC is written in C, while our implementation is executed by the JVM. We expect that if our implementation were optimized and/or coded in C, it would offer performance competitive with these incomplete systems while retaining completeness.

A factor of 8-20 speedup over Java for a C implementation is reasonable and supported by the results of [18]. The authors have shown that a C++ implementation of the Curry2Java abstract machine was more than 50 times faster than the same implementation in Java. We do not expect a similar improvement because we have already eliminated the interpretation layer of the abstract machine, and because the results of [18] were obtained with JDK 1.1 while we use JDK 1.3. The latter is more efficient. However, we are confident that there are still considerable opportunities for improving the efficiency of our implementation. We plan to work on this aspect, but only after resolving the architectural issues related to the inclusion of encapsulated search, which is a very interesting feature for modern functional logic languages [19]. Its integration could cause some changes in our backend, e.g., to distinguish between local and global variables, which is one important issue of encapsulated search. However, most of the structures needed (nested computations, fair scheduling, explicit control of computations etc.) are already available in our backend. Thus, we expect the integration of encapsulated search to cause only minor changes or extensions.

6 Conclusion

We described the architecture of an engine for functional logic computations. Our engine implements an efficient, sound and complete narrowing strategy, *INS*, and integrates this strategy with other features, e.g., residuation and and-parallelism, desirable in functional logic programming. Our implementation is operationally complete, easy to extend (e.g., by external resources like constraint libraries) and general enough to be used as a back end for a variety of languages. Although our work is still evolving, simple benchmarks show that it is the fastest complete implementation of narrowing currently available: it has strong potential for further improvement in both performance and functionality.

Our implementation and supporting material is available under the GNU Public License at `http://nmind.cs.pdx.edu`.

References

1. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.
2. E. Albert, M. Hanus, and G. Vidal. A practical partial evaluator for a multi-paradigm declarative language. In *Proc. 5th Intl. Symposium on Functional and Logic Programming (FLOPS '01)*, pages 326–342. Springer LNCS 2024, 2001.
3. S. Antoy. Definitional trees. In *Proc. 3rd Intl. Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
4. S. Antoy. Needed narrowing in Prolog. Technical report 96-2, Portland State University, 1996.
5. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. Intl. Conference on Algebraic and Logic Programming (ALP '97)*, pages 16–30. Springer LNCS 1298, 1997.
6. S. Antoy. Constructor-based conditional narrowing. In *Principles and Practice of Declarative Programming, (PPDP'01)*. ACM Press, Sept. 2001.
7. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal ACM*, 47(4):776–822, 2000. Previous version in *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, 1994.
8. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. 3rd Intl. Workshop on Frontiers of Combining Systems (FroCoS '00)*, pages 171–185. Springer LNCS 1794, 2000.
9. S. Bonnier and J. Maluszynski. Towards a clean amalgamation of logic programs with external procedures. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 311–326. MIT Press, 1988.
10. M. Chakravarty and H. Lock. Towards the uniform implementation of declarative languages. *Computer Languages*, 23(2-4):121–160, 1997.

11. P. Cheong and L. Fribourg. Implementation of narrowing: The Prolog-based approach. In K. Apt, J. de Bakker, and J. Rutten, editors, *Logic programming languages: constraints, functions, and objects*, pages 1–20. MIT Press, 1993.
12. E. Denti, A. Omicini, and A. Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In *Practical Aspects of Declarative Languages (PADL)*, pages 184–198. Springer LNCS 1990, 2001.
13. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
14. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth Intl. Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.
15. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
16. M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.3: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2000. Available at `http://www.informatik.uni-kiel.de/~pakcs`.
17. M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
18. M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, 1999(6), 1999.
19. M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint Intl. Symposium PLILP/ALP '98)*, pages 374–390. Springer LNCS 1490, 1998.
20. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.
21. T. Hortala-Gonzalez and E. Ullan. An abstract machine based system for a lazy narrowing calculus. In *Proc. 5th Intl. Symposium on Functional and Logic Programming (FLOPS '01)*, pages 216–232. Springer LNCS 2024, 2001.
22. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–443. MIT Press, 1991.
23. J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3):1–49, 1999.
24. R. Loogen. Relating the implementation techniques of functional and functional logic languages. *New Generation Computing*, 11:179–215, 1993.
25. R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. 5th Intl. Symposium on Programming Language Implementation and Logic Programming*, pages 184–200. Springer LNCS 714, 1993.
26. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proceedings of RTA '99*, pages 244–247. Springer LNCS 1631, 1999.
27. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji Intl. Symposium on Functional and Logic Programming (FLOPS '99)*, pages 100–113. Springer LNCS 1722, 1999.
28. A. Middeldorp. Call by need computations to root-stable form. In *Proc. 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 94–105, 1997.
29. J. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.
30. P. Tarau. Jinni. Available at http://www.binnetcorp.com/Jinni/, 2001.

Techniken für Preisvergleiche im World Wide Web

Stefan Kuhlins

Universität Mannheim
Lehrstuhl für Wirtschaftsinformatik III
L 5, 6
68131 Mannheim
stefan@kuhlins.de

Zusammenfassung Im Zuge der stürmischen Entwicklung des Electronic Commerce haben sich unzählige Online-Shops im World Wide Web angesiedelt, um dort ihre Produkte anzubieten. Zum günstigen Erwerb eines Produktes ist es generell sinnvoll, Preisvergleiche anzustellen. Dazu können Programme – beispielsweise Web-Robots und intelligente Softwareagenten – eingesetzt werden, die Preisinformationen zu Produkten von mehreren Online-Shops einholen und dann eine nach Preisen sortierte Übersicht erstellen.
In diesem Beitrag werden nach einer einführenden Motivation für Preisvergleiche im WWW, einige Techniken zur Realisierung vorgestellt. Es wird gezeigt, dass Preisvergleiche nur eine spezielle Anwendung dieser Techniken darstellen, die sich für zahlreiche weitere Anwendungen, welche Informationen mehrerer Web-Sites zusammenfassend auswerten, eignen. Des Weiteren soll ein Überblick über derzeit aktive Preisvergleichsangebote gegeben werden.

1 Einleitung

Für homogene Produkte („*Commodity Products*“) wie beispielsweise Musik-CDs, die sich nur durch den Preis unterscheiden, bietet Electronic Commerce prinzipiell ideale Möglichkeiten die Markttransparenz durch automatische Preisvergleiche zu erhöhen. Denn die Daten, die Online-Shops für ihre Kunden im WWW bereitstellen, sind maschinenlesbar und damit einer automatischen Bearbeitung zugänglich. Allerdings *verstehen* Maschinen den Inhalt nicht.

In der Regel werden Preisvergleiche im WWW dadurch realisiert, dass Preisinformationen mit Hilfe von so genannten *Wrappern* mühsam aus HTML-Seiten von Online-Shops extrahiert werden. Im Allgemeinen benötigt man dazu für jeden Online-Shop einen spezifischen Wrapper. Wenn sich das Layout der Web-Site ändert, ist der zugehörige Wrapper anzupassen.

Die Qualität der so extrahierten Preisinformationen gibt oft Anlass zur Kritik. Deshalb gehen immer mehr Online-Shops dazu über, ihre Preisinformationen zusammen mit Angaben über den angebotenen Service (Lieferbarkeit, Transportkosten usw.) zum Abruf bereitzustellen. Da sich bisher jedoch kein Standardformat dafür etabliert hat, ist dies erst ein kleiner Schritt in die richtige Richtung,

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

der es bekannten Geschäftspartnern erlaubt, ihre Daten auszutauschen. Für eine vollständige Automatisierung des Vorgangs werden standardisierte Schnittstellen benötigt, so dass die relevanten Daten und Funktionen in einheitlicher Weise vorliegen.

Dieser Beitrag ist folgendermaßen strukturiert: Kapitel 2 beschäftigt sich mit der Motivation für Preisvergleiche im WWW. Im dritten Kapitel wird der erste Preisvergleichsagent vorgestellt. Anschließend wird die zugrunde liegende Technik in Kapitel 4 erläutert. Ein praktischer Test heute aktiver Preisvergleichsagenten ist Gegenstand des fünften Kapitels. Abschließend folgen Zusammenfassung und Ausblick.

2 Motivation für Preisvergleiche im WWW

In einem Special Report von *Ernst & Young* [5], der auf einer im Oktober/November 2000 durchgeführten Online-Umfrage, an der 7.222 Konsumenten aus zwölf Ländern teilnahmen, basiert, heißt es: *"Commodity products are still the top sellers."* Beispielsweise haben von den befragten Deutschen 70 % Bücher, 60 % Computerprodukte und 53 % Software- sowie Musik-CDs mindestens einmal im WWW eingekauft.

Als Hauptgrund für einen Online-Einkauf wurde das Sparen von Zeit und Geld angegeben, weil für die Kunden der Weg ins Geschäft entfällt und weil die Unternehmen aufgrund elektronischer Bestellungen Abwicklungskosten sparen, was bei den Kunden wiederum die Erwartung nach günstigeren Konditionen im Vergleich zum physischen Einkauf im Geschäft weckt. Als wichtigste Ursachen für den Abbruch von Einkaufstransaktionen nannten die Befragten, dass die Lieferkosten zu hoch seien (40 %), dass der Preis zu hoch sei (35 %) oder dass sie den Preis nur prüfen wollten (33 %).

Commodity Products unterscheiden sich per definitionem nur im Preis. Daher können Preisvergleiche für sie verhältnismäßig einfach realisiert werden; für Handy-Tarife beispielsweise ist dies schwieriger. Da *Commodity Products* die beliebtesten Produkte für den Einkauf im WWW darstellen, besteht potenziell eine große Nachfrage nach Preisvergleichen für sie. Die Preissensibilität der Online-Kunden verstärkt diese Nachfrage zusätzlich.

Der hohe Stellenwert der Lieferkosten beim Abbruch von Einkaufstransaktionen zeigt, dass der Produktpreis allein nicht entscheidend ist. Es sollten möglichst alle mit einem Kauf verbundenen Kosten berücksichtigt werden.

Mit einem Browser können unzählige Online-Shops im WWW bequem vom heimischen PC aus besucht werden. Für ein begehrtes Produkt können so in verhältnismäßig kurzer Zeit viele Angebote eingeholt werden. Verglichen mit der Situation vor Einführung des Electronic Commerce sinken dadurch die Kosten für die Suche nach einem günstigen Anbieter. Aber auch das Finden möglichst vieler Online-Shops, das Laden ihrer Seiten und die Suche nach dem gewünschten Produkt sind mit einem erheblichen Zeitaufwand und entsprechenden Kosten verbunden. Diese stumpfsinnige Tätigkeit sollte daher automatisiert werden.

3 Der erste Preisvergleichsagent

Der erste Softwareagent für automatisierte Preisvergleiche war *BargainFinder*, der am 30. Juni 1995 der Öffentlichkeit vorgestellt wurde [7]. *BargainFinder* konnte für Musik-CDs, die durch Interpret und Titel identifiziert werden, bei rund einem Dutzend Anbietern – darunter *CD Universe*, *CDNow!*, *CD Land* und *CDworld* – die Preise ermitteln.

Einige Händler waren mit den extrahierten Daten unzufrieden, weil der Service, den sie ihren Kunden boten, nicht gewürdigt wurde. Im reinen Preisvergleich schnitten sie beispielsweise schlechter ab, weil Kosten für die Lagerhaltung zur Erzielung kurzer Lieferzeiten zu höheren Preisen führen. Aus diesem Grund wurde *BargainFinder* von einigen Händlern (u. a. *CDNow!* und *CD Land*) boykottiert. Andere Händler dagegen baten darum, berücksichtigt zu werden; und auch *CD Land* hob die Blockade später wieder auf.

Technisch konnte *BargainFinder* gezielt boykottiert werden, weil die Anfragen immer von demselben Server (mit derselben IP-Adresse) ausgingen. Bei einer Client-basierten Realisierung würden die Anfragen demgegenüber vom Kunden-PC ausgehen und wären von den üblichen Kundenanfragen nicht zu unterscheiden. Aus technischer Sicht haben Online-Shops daher keine Möglichkeit, automatisierte Preisvergleiche zu verhindern, ohne gleichzeitig ihre potenzielle Kundschaft zu treffen.

4 Die zugrunde liegende Technik

Die Informationen, die Online-Shops für ihre Kunden bereitstellen, liegen in maschinenlesbarer Form vor, damit Browser sie anzeigen können. Deshalb ist es prinzipiell möglich, Preise automatisch zu extrahieren. Allerdings werden Online-Shops für die Benutzung durch Menschen entworfen. Insbesondere HTML-Seiten bieten zwar ein nett anzusehendes Layout, stellen aber kaum maschinenverwertbare Informationen darüber zur Verfügung, was die dargestellten Daten bedeuten. Mit welchen Problemen Softwareagenten wie *BargainFinder* zu kämpfen haben, soll das folgende Beispiel, Preisermittlung für die Digitalkamera *Sony DSC-P1* bei *Imaging One* [6], verdeutlichen.

Zunächst muss innerhalb eines Online-Shops die passende Seite zu einem Produkt gefunden werden. Kleine Shops offerieren ihre Produkte oft nur auf einer einzigen Seite. Größere Shops benutzen hingegen in der Regel einen hierarchisch aufgebauten Produktkatalog. Für einen schnelleren Zugang gibt es meistens Suchformulare. Diese stellen auch einen guten Einstiegspunkt für Softwareagenten dar.

Wenn die Seite mit den Produktinformationen gefunden wurde, sind die Preisinformationen zu extrahieren. Bei *Imaging One* handelt es sich um eine HTML-Seite, die aus mehreren geschachtelten Tabellen besteht (siehe Abb. 1). Kamerabezeichnung, Lieferbarkeit und der Preis sind für das menschliche Auge übersichtlich angeordnet und leicht zu erfassen. Der zugehörige HTML-Code mutet indessen chaotisch an (siehe Abb. 2). Die gesamte HTML-Datei umfasst

Abbildung 1. Ausschnitt der Browseransicht bei *Imaging One* [6]

625 Zeilen, deshalb wird in Abb. 2 nur der für die Preisermittlung relevante Ausschnitt abgebildet.

Das dargestellte Codefragment enthält einen Fehler: Ein schließendes Font-Tag ist überflüssig. Für die komplette Datei meldet *HTML Tidy* [8] 125 Fehler und Warnungen. HTML-Browser sind sehr tolerant, was die Darstellung von fehlerhaften HTML-Seiten angeht. Infolgedessen müssen Softwareagenten ebenfalls fehlertolerant programmiert werden, wodurch sich der Implementierungsaufwand erhöht.

Zum Extrahieren von Informationen aus HTML-Dateien kann man reguläre Suchausdrücke verwenden. Für das Beispiel in Abb. 2 liefert der reguläre Ausdruck `<a name="\([0–9]+\)"></a>.*<br>\n *\(.*\)</strong>` in `\1` (erster geklammerter Ausdruck) die Bestellnummer und in `\2` die Artikelbezeichnung. Mit ähnlichen Ausdrücken können Preis und Lieferbarkeit ermittelt werden. Der erste Treffer enthält die Daten der Digitalkamera. Nachfolgende Treffer gibt es für das Zubehör.

Eine weitere Möglichkeit zum Auffinden der gewünschten Daten besteht in der Nutzung des *Document Object Model* (DOM) [10]. Damit kann in der Baumstruktur eines HTML-Dokuments navigiert werden, um auf einzelne Elemente zuzugreifen. Im Beispiel ist die Bestellnummer durch `html.body.table[0].table[1].tr[0].td[0].table[0].tr[0].td[1].table[0].tr[1].td[1].strong[0].font[0].a[0]:name` und die Artikelbezeichnung durch `...strong[0].PCDATA[0]` gekennzeichnet. Dabei ist die Notation an W4F (*WysiWyg Web Wrapper Factory*, ein Toolkit zur Wrapper-Generierung) angelehnt [9].

Würden die Online-Shops XML an Stelle von HTML zur Gestaltung ihrer Web-Sites einsetzen, wäre es leichter, die gewünschten Informationen zu extra-

```
<html>
...
 </tr>   <tr>
  <td width="5" bgcolor="#669999"> </td>
  <td valign="top"><strong><font size="2"><a name="101150"></a> </font><br>
   Sony Digitalkamera DSC-P1</strong> 
    <font color="#008080" size="2">ab Lager lieferbar</font><br>
   Kamera inkl. InfoLithium-Akku NP-FS11, Netz-/Ladegerät, 8 MByte Memory Stick, ...
<br>- Auflösung 2.048 x 1.536 Bildpunkte
<br>- CCD-Sensor mit 3.340.000 Pixeln
<br>- 3-fach Zoomobjektiv 39 bis 117 mm (entspr. Kleinbild)
<br>- eingebauter 1,5"-LCD-Monitor
<br>- Videoausgang (PAL/NTSC umschaltbar)
<br>- USB-Schnittstelle
<br>- Memory Stick Wechselspeicher
<br>- 12 Monate Herstellergarantie
<br><a href="../Kameras/DB/SonyDSC-P1.htm"><font size="2">ausführliche
    Technische Daten</font></a></font></td>
  <td width="5"></td>
  <td valign="top" width="90" align="center"><font size="2"> </font><br>
   101 150</td>
  <td align="right" valign="top" width="80"><font size="2"> </font><br>
   1.829,- </td>
  <td width="5" bgcolor="#000000"> </td>
 </tr>   <tr>
...
</html>
```

Abbildung 2. Ausschnitt des HTML-Codes

hieren, weil XML eine Trennung von Inhalt und Layout vorsieht. (Dieser Effekt lässt sich unter Verwendung von Style-Sheets zum Teil auch für HTML erzielen.) Da in XML-Dateien eigene Tags zur Beschreibung des Inhalts gewählt werden können, wäre ein Durchbruch aber erst erzielt, wenn alle Online-Shops dieselbe DTD benutzen würden. So oder so dürfte es noch einige Jahre dauern, bis sich XML auf breiter Front durchgesetzt hat; und selbst dann wird man HTML-Altlasten nicht völlig ignorieren können.

Softwaremodule, die Daten aus einer speziellen Quelle, beispielsweise den HTML-Seiten eines Online-Shops, extrahieren und Datenstrukturen zur weiteren Verarbeitung liefern, werden *Wrapper* genannt. Mit Hilfe der Wrapper-Technik (siehe Abb. 3) können Daten aus unterschiedlichen Ressourcen abgefragt und durch einen *Mediator* integriert werden [11].

Die Wrapper für *BargainFinder* waren handcodiert. Das größte Problem handcodierter Wrapper ist der enorme Aufwand zur Entwicklung und Wartung. Da jeder Online-Shop sein eigenes Layout besitzt, braucht man für jeden Shop einen speziellen Wrapper. Ändert sich das Layout eines Shops, ist auch der zugehörige Wrapper anzupassen. Der Einsatz von Werkzeugen zur Wrapper-Generierung (beispielsweise W4F [9]) beschleunigt zwar die Entwicklung von Wrappern, ändert aber am grundsätzlichen Problem nichts.

Mit *ShopBot* wurde demzufolge versucht, ein System zu entwickeln, das automatisch Wrapper zum Extrahieren von Preisen generiert [4]. Dazu verfügte *ShopBot* über einen Lernalgorithmus, der typische Eigenschaften von Online-Shops, wie das Vorhandensein von Formularen zur Suche nach Produkten und die

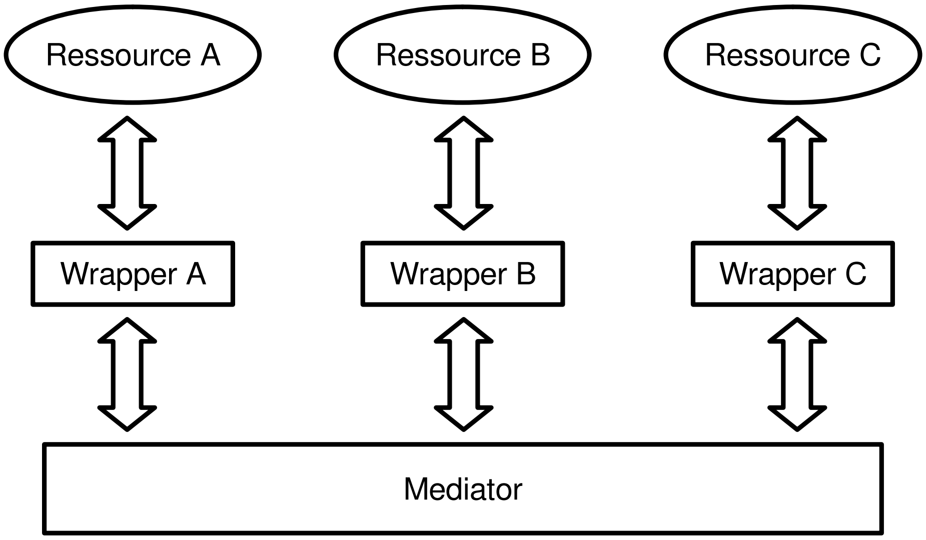


Abbildung 3. Wrapper-Technik

zeilenorientierte Präsentation von Produkten, ausnutzte. Obwohl erste Tests ansprechende Ergebnisse lieferten, wurde eine vollständige Automatisierung nicht erreicht, was u. a. durch folgende Fußnote zum Ausdruck gebracht wurde [4]: "*Prices are extracted using special hand-coded techniques.*"

BargainFinder und *ShopBot* sind die einzigen Softwareagenten für Preisvergleiche, deren Arbeitsweise in öffentlich zugänglichen Artikeln dokumentiert wurde. Die Arbeitsweise heute aktiver Systeme wird aus Wettbewerbsgründen nicht offen gelegt. Eine Liste mit deutschen und internationalen Anbietern für Preisvergleiche befindet sich unter [1].

5 Heute aktive Preisvergleichsagenten

Ein am 1. März 2001 in den USA durchgeführter praktischer Test zeigt, dass sich mit Hilfe von Preisvergleichen viel Geld sparen lässt. Für die Digitalkamera *DSC-P1* gibt der Hersteller *Sony* eine unverbindliche Preisempfehlung von $ 799,95 an. Zu diesem Preis wird das Produkt von einigen Händlern auch angeboten. Preisvergleiche mit *DealTime*, *mySimon*, *BizRate*, *GoTo Shopping*, *PriceGrabber* und *PriceSCAN* bringen dagegen Preise von unter $ 640 hervor, was einer Ersparnis von rund $ 160 oder 20 % entspricht (siehe Tab. 1). Die zweite Spalte der Tabelle 1 enthält die Anzahl der Angebote („Treffer"), die beim Preisvergleich aufgelistet werden. In der dritten Spalte steht der günstigste Preis. Sofern man die Qualität der Anbieter von Preisvergleichen nur nach dem von ihnen aufgespürten günstigsten Preis bewertet, unterscheiden sie sich mit einer maximalen Preisdifferenz von $ 2 kaum nennenswert.

Für die Auswahl eines Händlers, bei dem ein Produkt erworben werden soll, sind neben dem Preis weitere Faktoren von Bedeutung. Bei einigen Preisvergleichen werden deshalb die Lieferbarkeit (vierte Spalte in Tab. 1), Lieferkosten

	Anzahl	$-Preis	Lieferbar	Lieferkosten	Rating
DealTime.com	39	637,99	X	– (selten)	** Gómez
mySimon.com	41	637,99	–	–	** Gómez
BizRate.com	19	638,00	X	–	8,2
GoTo Shopping	22	639,90	X	$ 0,25	–
PriceGrabber.com	44	639,99	X	ca. $ 38,40	Not Rated
PriceSCAN.com	36	639,99	See Site	See Site	–

Tabelle 1. Preisvergleich in den USA

und Händler-Ratings, die gewisse Rückschlüsse auf die Qualität von Händlern ermöglichen sollen, angegeben. Während die Lieferbarkeit immerhin bei vier von sechs Preisvergleichen aufgeführt wird, fehlen Angaben über die Lieferkosten weitestgehend. Vermutlich ist dies darauf zurückzuführen, dass der Aufwand zum Extrahieren dieser Informationen den der einfachen Preissuche noch erheblich übersteigt. Die Kunden müssen diese Daten deshalb selbst recherchieren.

Für Deutschland ergeben sich ähnliche Ergebnisse. So brachte ein Preisvergleich mit *DealTime Deutschland* [3] einen Preis von 1.574 DM bei insgesamt 36 Treffern hervor (siehe Abb. 4). Wieder beträgt die Ersparnis gegenüber Sonys unverbindlicher Preisempfehlung von 1.999 DM rund 20 %. Angaben über Lieferbarkeit und Lieferkosten fehlen allerdings ebenso wie ein Händler-Rating.

Interessanterweise kostet die Kamera beim günstigsten Händler in Deutschland umgerechnet um $ 745, also über $ 100 mehr als in den USA. Diese Preisdifferenz von rund 15 % ist aber wohl nicht groß genug, damit sich in diesem Fall ein Einkauf bei einem amerikanischem Händler rentiert. Trotzdem zeigt das Beispiel, das Preisvergleiche über Ländergrenzen hinweg sinnvoll sein können, wenn dabei Wechselkurse, Transportkosten, Zollgebühren usw. eingerechnet werden.

6 Zusammenfassung und Ausblick

Die genannten kommerziellen Systeme für Preisvergleiche sind aus Wettbewerbsgründen kaum öffentlich dokumentiert. Es ist aber davon auszugehen, dass viele auf Web-Robots und Wrappern basieren, also ähnlich wie *BargainFinder* arbeiten. Wenn man sich vor Augen hält, dass die meisten Online-Shops ihre Produktdaten in Datenbanken speichern und Eingabeformulare zur Suche nach bestimmten Produkten anbieten, wobei als Antwort auf derartige Abfragen HTML-Seiten mit den Datenbankinhalten automatisch generiert werden, dann dienen Wrapper, die diese Produktdaten aus den generierten HTML-Seiten extrahieren, einer Art Reverse-Engineering von Datenbankschnittstellen in HTML. Insgesamt handelt es sich also um einen äußerst ineffizienten Prozess.

Es gibt Online-Shops, die mit Anbietern von Preisvergleichen kooperieren, indem sie ihre Produkt- und Preisdaten auszugsweise (in komprimierter Form) zum Download zur Verfügung stellen, wobei der von ihnen gebotene Kundenservice Berücksichtigung finden kann. Dazu werden in der Regel jedoch individuelle Formate verwendet, so dass auch hier geeignete Wrapper zur Verarbeitung benötigt werden. Allerdings ist die Erstellung solcher Wrapper einfacher und der Wartungsaufwand entfällt weitestgehend, weil eine Änderung des Layouts der Web-Site sich nicht auf das Format der Datei mit den Produktdaten auswirkt. Mit steigendem Produktangebot nimmt auch die Dateigröße zu, infolgedessen ist diese Art der Datenbereitstellung nur für kleine Online-Shops geeignet.

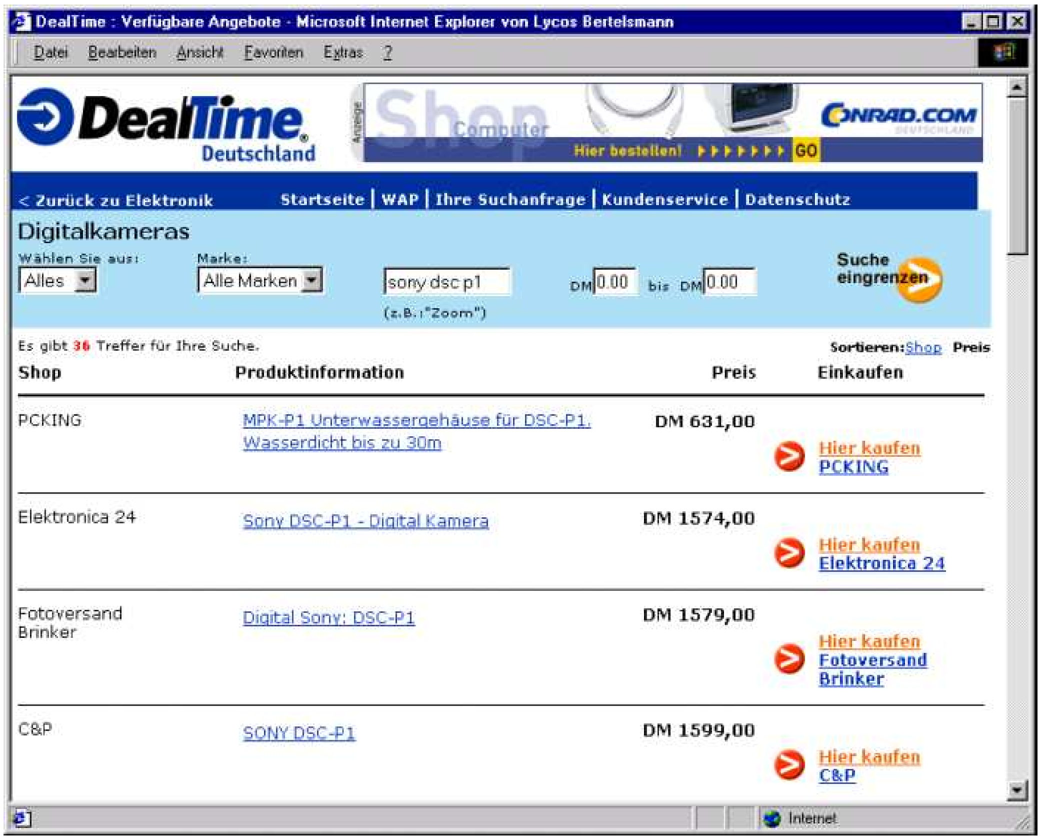


Abbildung 4. DealTime Deutschland [3]

Wenn Online-Shops kooperieren, eröffnen sich aber auch bessere Möglichkeiten als ein Dateidownload. Zur Ermittlung von Produkt- und Preisdaten wäre es am besten, wenn Online-Shops eine einheitliche Schnittstelle für Abfragen zur Verfügung stellen würden, wobei die Informationen zum jeweils gebotenen Kundenservice berücksichtigt würden. Dann könnten Softwareagenten für ein bestimmtes Produkt alle für einen Kauf relevanten Daten analog zu einfachen Datenbankabfragen ermitteln. In Verbindung mit einem Verzeichnis, in dem sämtliche Online-Shops aufgeführt werden, könnten so umfassende Preisübersichten erstellt werden.

Für den direkten Weg zu den Produkt- und Preisdaten wäre es allerdings unumgänglich, dass Online-Shops geeignete Schnittstellen für entsprechende Abfragemöglichkeiten installieren. Wenn eines Tages die meisten Kunden Preisvergleiche im WWW nutzen, könnten die Shops dadurch faktisch gezwungen sein, mitzumachen, weil die Umsätze nicht gelisteter Shops rapide zurückgehen. Um einem ruinösen Preiswettbewerb zu entrinnen, werden die Shops in Bezug auf *Commodity Products* versuchen, sich über den Kundenservice zu differenzieren, was letztendlich auch im Interesse der Kunden ist. Die Kunden können dann selbst entscheiden, wie viel ihnen beispielsweise eine kürzere Lieferzeit wert ist.

Bisher decken die Anbieter von Preisvergleichen jeweils nur einen Teilbereich des relevanten Marktes ab. Analog zu Meta-Suchmaschinen, die mehrere Suchmaschinen abfragen und deren Ergebnisse zusammenfassen, wäre daher ein Meta-Preisvergleich denkbar. Ob die einzelnen Anbieter damit einverstanden sind, dürfte von ihrem Finanzierungsmodell abhängen. Außerdem wäre zu klären, ob ein Meta-Preisvergleich rechtlich überhaupt zulässig ist, sofern einzelne Anbieter, die abgefragt werden, damit nicht einverstanden sind.

Einige Anbieter von Preisvergleichen unterstützen die Abfrage mittels mobiler Geräte (z. B. Handys) unabhängig vom Aufenthaltsort. Die Verhandlungsposition der Kunden im „Geschäft um die Ecke“ wird damit erheblich gestärkt. Ein erster Vertreter dieser Art mobiler Preisvergleiche ist *Pocket BargainFinder* [2].

Zusammenfassend kann festgehalten werden, dass Preisvergleiche im WWW ein interessantes Anwendungsgebiet für Softwareagenten darstellen und dass die zugrunde liegenden Techniken für zahlreiche Anwendungen, bei denen Informationen aus mehreren Quellen integriert werden, nutzbar sind.

Literatur

1. Anbieter von Preisvergleichen im WWW, http://www.wifo.uni-mannheim.de/~kuhlins/preisvergleich/
2. Brody, Adam B. und Edward J. Gottsmann: Pocket BargainFinder – A Handheld Device for Augmented Commerce, in: Gellersen, Hans-W. (Hrsg.): Handheld and Ubiquitous Computing, Springer-Verlag, Berlin 1999, S. 44–51, http://www.accenture.com/xdoc/en/services/cstar/publications/PocketBargainFinder-HUC99.PDF
3. DealTime Deutschland: http://www.dealtime.de/
4. Doorenbos, Robert B., Oren Etzioni und Daniel S. Weld: A scalable comparison-shopping agent for the World-Wide Web, in: Proceedings of the International Conference on Autonomous Agents, Marina del Rey, CA, 1997, S. 39–48, ftp://ftp.cs.washington.edu/pub/etzioni/softbots/agents97.ps
5. Ernst & Young: Global Online Retailing 2001, Special Report, http://www.ey.com/global/vault.nsf/US/2001_Retail_Study/$file/GOR.pdf
6. Imaging One: Digitalkamera *Sony DSC-P1*, http://www.imaging-one.de/Kameras/SonyDSC-P1.htm (abgerufen am 1.3.2001)
7. Krulwich, Bruce: Bruce Krulwich's ideas page, http://www.geocities.com/ResearchTriangle/9430/
8. Raggett, Dave: HTML Tidy, 30. April 2000, http://www.w3.org/People/Raggett/tidy/

9. Sahuguet, Arnaud und Fabien Azavant: W4F (WysiWyg Web Wrapper Factory), http://db.cis.upenn.edu/W4F/
10. W3C: Document Object Model (DOM), http://www.w3.org/DOM/
11. Wiederhold, Gio: Mediators in the Architecture of Future Information Systems, IEEE Computer, 25(3):38–49, 1992, http://www-db.stanford.edu/pub/gio/1991/afis.ps

Zur formalen Beschreibung von Aminosäureketten und Proteinen

Hermann von Issendorff
Institut für Netzwerkprogrammierung
D-21745 Hemmoor, Hauptstrasse 40
hviss@issendorff.de

Zusammenfassung. Proteine haben nicht nur eine räumliche Struktur, sondern sind auch in der Lage, bestimmte Funktionen auszuführen. Andererseits wird ein Proteine als Kette von Aminosäuren aufgebaut, die sich nach der Fertigstellung zu einem Protein zusammenzieht. In der Kette von Aminosäuren müssen daher bereits alle Eigenschaften des Proteins codiert sein. Die Aminosäurenkette kann somit auch als ein in einer bisher unbekannten Proteinsprache geschriebenes Programm gedeutet werden. In diesem Beitrag stellen wir eine formale Methode vor, Aktonalgebra genannt, mit der sich gleichermassen räumliche Strukturen und DV-Funktionen beschreiben lassen. Sie vermag vielleicht dazu beizutragen, die Proteinsprache schneller zu entschlüsseln.

1 Einleitung

Jede Zelle eines Organismus ist eine kleine Fabrik, in der u.a. fortwährend bestimmte Proteine erzeugt und verbraucht werden. Die bei der Proteinerzeugung ablaufenden Vorgänge sind recht genau bekannt. Ribosome suchen zu diesem Zweck bestimmte Abschnitte auf den DNA-Strängen, lesen den darin enthaltenen Gencode und erzeugen daraus eine Kette von Aminosäuren. Jede 3-er Folge von Nukleinsäuren beschreibt genau eine von 20 verschiedenen Aminosäuren.

Sobald die Aminosäurenkette freigegeben wird, zieht sie sich zusammen und bildet so ein Protein. Das Zusammenziehen geschieht mit grosser Geschwindigkeit und vor allem mit grosser Präzision und führt zu einer bestimmten räumlichen Struktur. Abhängig von der Reihenfolge der Aminosäuren und der Kettenlänge kann auf diese Weise eine enorme Zahl verschiedener Proteine gebildet werden.

Der eigentliche Zweck der verschiedenen Proteine besteht in den Funktionen, die sie innerhalb der Zelle ausüben. Die erwähnten Ribosome z.B. sind Proteine, die die Fähigkeit haben, auf einem DNA-Strang entlang zu wandern, dem Gencode entsprechend bestimmte Aminosäuren aus der Umgebung aufzunehmen und aneinander zu ketten. Andere Proteine, Proteasen genannt, betreiben z.B. so etwas wie Qualitätssicherung. Sie prüfen die Korrektheit der Proteine und zerschneiden sie in Einzelstücke, wenn sie fehlerhaft sind.

Die Struktur und die Funktionen eines Proteins hängen auf das Engste zusammen. Eine bestimmte Funktion kann nur ausgelöst werden, wenn ein anderes Molekül, etwa ein anderes Protein, mit seinen lokalen chemischen Bindungen genau zu dem Protein passt. Eine gängige Vorstellung ist die vom Schlüssel und dem Schloss.

Die 20 Aminosäuren, aus denen die Proteine gebildet werden, lassen sich als die Elemente einer formalen Sprache interpretieren, mit der sich nicht nur Funktionen beschreiben lassen, wie das von den konventionellen Programmiersprachen bekannt ist,

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

sondern auch räumliche Strukturen. Eine Kette von Aminosäuren ist unter dieser Interpretation ein Programm, das eine reale Maschine beschreibt, die in Wechselwirkung mit ihrer Umgebung bestimmte Funktionen ausführen kann. Eine Programmiersprache, die diese Eigenschaften hat, ist bisher nicht bekannt.

Wir wollen uns in dieser Arbeit mit genau dieser Frage befassen. Dabei wird zunächst die Frage nach der formalen Beschreibung räumlicher Strukturen im Vordergrund stehen. "Beschreibung" bedeutet dabei "Darstellung als (eindimensionaler) String". Eine derartige Sprache muss aber ausserdem die Eigenschaft haben, die auf oder in den Proteinen ablaufenden Vorgänge zu beschreiben.

Die zur Beschreibung der Strukturen und Vorgänge verwendete Sprache ist die Akton-Algebra, abgekürzt AA, über deren Vorstufen in diesem Workshop schon verschiedentlich berichtet wurde. Bei geeigneter semantischer Interpretation lassen sich mit der AA auch die räumlichen Strukturen von Biomolekülen beschreiben, u.a. auch Helices und Faltungen, die für Proteine typisch sind. Die AA benötigt dafür insgesamt 6 Grundsorten von strukturbildenden Aktonen und dazu mindestens eine Grundsorte funktioneller Aktonen. Die Natur verwendet für den gleichen Zweck 20 verschiedene Aminosäuren. Es besteht damit eine gewisse Aussicht, dass sich die strukturellen und funktionellen Eigenschaften der verschiedenen Aminosäuren als Akton-terme beschreiben lassen. Zumindest könnte die AA einen Hinweis zur Entschlüsselung des Aminosäurencodes geben.

Auf dem Gebiet der klassischen Informatik sind bisher nur wenige Versuche bekannt, die räumliche Struktur von Systemen formal zu beschreiben, und diese sind sehr speziell. Beispiele dafür sind Ruby [1] und CADIC [2]. Beide Ansätze wurden für die Darstellung planarer Schaltungen entwickelt. Ruby ist auf die Beschreibung kombinatorischer Digitalschaltungen beschränkt, und damit auf gerichtete, azyklische, planare Strukturen. CADIC wurde für Layout-Zwecke entwickelt und beschreibt lediglich ungerichtete, planare Strukturen. Auf dem Gebiet der Bioinformatik gibt es Ansätze, planare Strukturen formal zu beschreiben, wie sie z.B. bei Faltungen auftreten [3, 4]. Diese Ansätze enthalten aber keinen Hinweis darauf, wie sie auf die Beschreibung räumlicher Strukturen oder gar Proteinfunktionen erweitert werden könnten.

2 Akton-Algebra

AA beschreibt ein physikalisches System, wie es von einem Beobachter gesehen wird, der zwischen rechts/links, oben/unten und vorne/hinten unterscheidet. Die Komponenten des Systems werden als gerichtet angenommen, d.h. Input und Output liegen auf entgegengesetzten Seiten. Dies ist keine Beschränkung, da Teilstrukturen mit mehreren Inputs und Outputs stets in Komponenten zerlegt werden können. Der Beobachter nimmt immer eine Position ein, sodass abhängige benachbarte Komponenten von links nach rechts angeordnet sind und nichtabhängige benachbarte Komponenten oben und unten. Bei Kreuzungen von Komponenten wird die hintere zerschnitten, und die Schnittenden werden markiert. Der Beobachter sieht damit ein physikalisches System ausgebreitet auf einer Ebene und von links nach rechts orientiert. Die originäre (dreidimensionale) Systemstruktur kann darin auf zwei verschiedene Weisen vollständig beschrieben werden, entweder durch Angabe der Raumwinkel zwischen verbundenen

Komponenten oder durch Annahme physikalischer oder chemischer Anziehungskräfte, wo immer die Systemstruktur in der Darstellung gedehnt oder zerschnitten ist.

AA ist eine mehrsortige Term-Algebra, deren Elemente Aktonen genannt werden. Ein Akton repräsentiert eine Systemkomponente als schwarzen Kasten, d.h. es zeigt nicht seine innere Struktur. Ein Akton hat einen geordneten Input und einen geordneten Output. Beide können leer sein oder eine beliebige (endliche) Menge von Elementen enthalten. Jedes Element hat eine eigene räumliche Position. Diese Position dient zur Identifikation des Elements und bildet das Kriterium für die Oben/Unten-Ordnung der Elemente im Input und Output. Wenn ein Akton Input- und Output-Elemente enthält, dann sind diese üblicherweise durch ein Verarbeitungsnetzwerk miteinander verbunden. Die Verarbeitung, d.h. die Erzeugung eines Output-Elementes aus einem oder mehreren Input-Elementen, dauert eine individuelle Zeit. Die Produktion verschiedener Output-Elemente eines Aktons ist üblicherweise nicht synchronisiert. Jedes Akton mit nichtleerem Output hat mindestens ein Akton als Nachfolger, und jedes Akton mit nichtleerem Eingang hat mindestens ein Akton als Vorgänger.

Ein Term repräsentiert ein Teilsystem als weissen Kasten, d.h. ein Term zeigt seine innere Struktur. Ansonsten gleicht ein Term einem Akton, d.h. es hat einen geordneten Input und einen geordneten Output. Ein Term besteht aus einem Akton oder jeder Aktonstruktur, die mit Hilfe von zwei binären Operatoren gebildet werden kann. Ein weisser Kasten kann immer zu einem schwarzen Kasten abstrahiert werden. Auf diese Weise können Aktonen beliebiger Grösse und Komplexität erzeugt werden.

Zwei benachbarte Terme x und y werden durch $x{:}y$ beschrieben, wenn der Output von x die gleichen Elemente wie der Input von y enthält. Der Doppelpunkt ist ein *Next* genannter binärer Operator. *Next* definiert - der Schreibrichtung entsprechend - eine Richtung von links nach rechts.

Zwei benachbarte Terme x und y, die keine Input/Output-Beziehung haben, werden durch x/y beschrieben, wenn x oberhalb von y liegt. Der Schrägstrich ist ein *Juxta* genannter binärer Operator. Beide Terme x und y teilen sich in den Input des Term x/y. Term x bezieht seinen Anteil vom oberen Ende und Ter y vom unteren Ende. Überlappen sich beide Anteile, wird dieser Bereich gegabelt.

Die durch *Next* und *Juxta* definierten Links/Rechts-Richtungen und Oben/Unten-Richtungen spannen die Beobachtungsebene auf.

Die Infix-Notation der beiden binären Operatoren dient lediglich der besseren Lesbarkeit. Sie erfordert jedoch die Einführung von Klammern zur Abgrenzung der Terme. Wie üblich wird die Klammerzahl dadurch reduziert, dass für *Juxta* eine stärkere algebraische Bindung angenommen wird als für *Next*. Für eine maschinelle Bearbeitung ist natürlich die klammerfreie Polnische Notation vorzuziehen.

Ausser den binären Operatoren gibt es zwei Typen von Aktonen, Strukuraktonen und Funktionsaktonen genannt.

Insgesamt gibt es 6 Strukturaktonen, die verschiedene Grundstrukturen darstellen. Als Erstes werden die komplemtentären Strukturaktonen *Entry* ($*$) und *Exit* ($*$') eingeführt. Sie dienen als Systembegrenzer. *Exit* exportiert Elemente in die sichtbare Umgebung und *Entry* importiert Elemente von dort. Demgemäss hat *Exit* einen leeren Output und einen nichtleeren Input. Bei *Entry* ist es umgekehrt. Wenn der Output von *Entry* dem Input von *Exit* gleicht und beide planar benachbart sind, wird das Paar passend (matching) genannt. Nichtpassende, planar benachbarte *Entries* und *Exits* werden durch unterschiedliche Indices unterschieden.

Ein nächstes Paar von Strukturaktonen wird *Up(*o*)* und *Down(*o'*)* genannt. *Up* hat die gleichen Eigenschaften wie *Entry* und *Down* wie *Exit*, ausser dass sie eine Verbindung zu einer Umgebung darstellen, die unterhalb der Beobachtungsebene liegt und damit nicht sichtbar ist. Ein passendes (matching) *Down/Up*-Paar ist daher räumlich benachbart. Wie später noch behandelt wird, lassen sich mit *Down/Up*-Paaren gerichtete Kreuzungen eindeutig beschreiben, einschliesslich des Drehsinns.

Ein weiteres Strukturakton wird *Link ($)* genannt. Sein Input und sein Output haben die gleiche Ordnung und die gleiche Kardinalität. Ein *Link* dient dazu, räumlich getrennte abhängige Aktonen miteinander zu verbinden und hat damit die Eigenschaft eines Verbindungskabels.

Das letzte Strukturakton wird *Gap* (#) genannt. Es hat einen leeren Input und einen leeren Output und importiert oder exportiert nichts. Es kann als ein Stück Materie betrachtet werden, das als Abstandshalter dient.

Die Strukturaktonen und die für ihre algebraische Darstellung verwendeten Symbole sind in der nachstehenden Tabelle zusammengefasst:

Entry	*Exit*	*Up*	*Down*	*Link*	*Gap*
*	*'	o	o'	*$*	*#*

Funktionsaktonen haben geordnete Inputs und Outputs. Jedes Output-Element ist über ein Verarbeitungsnetzwerk mit Input-Elementen und/oder aktoninternen Konstanten verbunden. Im Unterschied zum *Link* produziert ein Funktionsakton im Allgemeinen einen Output, der sich vom Input unterscheidet. Im einfachsten Fall besteht ein Funktionsakton aus einem digitalen Gatter oder einem Inverter. Der Output enthält in diesem Fall ein einziges Element. Funktionsaktonen sind aber nicht auf diese Schaltelemente beschränkt, sondern können Verarbeitungsnetzwerke jeder Grösse und Komplexität enthalten.

Eine Besonderheit der Funktionsaktonen ist, dass sie über eine explizit angegebene Bedingung kontrolliert werden können. Diese Bedingung ist eine dreiwertige Boolesche Variable, die konjunktiv auf alle Output-Elemente wirkt. Die drei Werte sind *1*, *0* und *u (undefiniert)*. Die Bedingung kann in eckigen Klammern an ein Funktionsakton angehängt werden. Das heisst, wenn *a* der Name eines Funktionsaktons ist und α die Bedingung, dann kann das bedingte Akton durch $a[\alpha]$ bezeichnet werden. Da üblicherweise Boolesche Variable verwendet werden, die nur die Werte *1* und *0* annehmen, führen wir eine spezielle Funktion *s* ein, die den Wert *0* in *u* überführt. Das heisst, wenn $p \in \{1, 0\}$ *ist*, dann ist $s(p) \in \{1, u\}$.

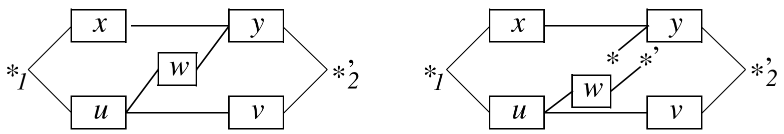
Wir verwenden in dieser Arbeit die Buchstaben *u,v,w,x,y,z* für Terme. Andere Buchstaben dienen als Bezeichner für Aktonen und andere Objekte.

3 Vom Raum zur AA

Die Beschreibung von physikalischen Systemen durch die AA beruht auf zwei wohldefinierten Abbildungen. Die erste Abbildung reduziert die drei Dimensionen des physikalischen Systems auf die zwei Dimensionen einer planaren Darstellung, und die zweite die planare Darstellung auf eine lineare. Die lineare Darstellung wird durch die AA wiedergegeben. Beide Abbildungen können durch ein Gummibandmodell visualisiert werden, in dem die Gummibänder die Input/Output-Beziehungen zwischen den Komponenten veranschaulichen. Dabei ist jedoch zu beachten, dass die Gummibänder nur virtuelle Beziehungen darstellen und sonst keine physikalische Bedeutung haben.

Das Gummibandmodell nimmt an, dass alle Verbindungen zwischen Aktonen beliebig gestreckt oder gebogen werden können. Die erste Abbildung beinhaltet die Ausrichtung aller Teilstrukturen von links nach rechts. Dies gilt insbesondere für topologische Schleifen und Rückkopplungszyklen. Das System wird so weit in Links/Rechts- und Oben/Unten-Richtung gedehnt, dass alle Aktonen für den Beobachter vollständig sichtbar werden. Dies ergibt ein System, das bezüglich der Aktonen planar ist, das aber noch lokale dreidimensionale Strukturen enthält, d.h. Kreuzungen, die von sich kreuzenden unabhängigen Verbindungen, topologischen Schleifen oder Rückkopplungszyklen stammen. Diese lokalen dreidimensionalen Strukturen können nun dadurch beseitigt werden, dass jede unterführende Verbindung zerschnitten wird und durch ein passendes *Down/Up*-Paar ersetzt wird. Das Endresultat dieser Abbildung ist eine gerichtete, partiell geordnete, planare Struktur.

Die zweite Abbildung dehnt die planare Struktur weiter in Links/Rechts-Richtung, bis alle Aktonen aufgereiht sind. Das geschieht dadurch, dass das oberste *Entry* oder *Up* an der linken Seite und das unterste *Exit* oder *Down* an der rechten Seite auseinander gezogen werden. Auf diese Weise lassen sich benachbarte unabhängige Teilstrukturen ineinander verschachteln (interleaving), und zwar so, dass obere Teilstrukturen vor den unteren liegen. Es gibt jedoch auch benachbarte Teilstrukturen, die untereinander verbunden und damit nicht unabhängig sind. Solche Teilstrukturen können nicht in eindeutiger Weise verschachtelt werden. Das Problem ist in der Abbildung 1 dargestellt.



Planare Aktonstruktur Modifizierte planare Aktonstruktur

$*_1:(x/*:y)/(u:(w:*')/v):*'_2$

Lineare Aktonstruktur

Abbildung 1. Nichtverschachtelbare Aktonstruktur, verschachtelbare Aktonstruktur und AA-Ausdruck

Die Darstellung der Aktonenstrukturen in Abbildung 1 und allen folgenden Abbildungen muss erläutert werden. Funktionsaktonen und Funktionsaktonen enthaltende Teilstrukturen werden durch benannte Kästchen dargestellt und Strukturaktonen durch ihre Symbole. Die Linien sind die Gummibänder. Durchgehende Linien kennzeichnen explizite Abhängigkeiten zwischen Teilstrukturen. Implizite Beziehungen passender *Exit/Entry*-Paare können durch gebrochene Linien angedeutet werden und die passender *Down/Up*-Paare durch gepunktete Linien.

Die planare Struktur links in Abbildung 1 besteht aus einer oberen Teilstruktur *x,y*, einer unteren *u,v* und einer sie verbindenden Teilstruktur *w*. Teilstruktur *w* hat die Eigenschaft, sowohl unter *x* zu liegen als auch über *v* und kann deshalb nicht eindeutig in eine Kette eingeordnet werden, dass sie vor *v* und hinter *x* liegt. In der planaren Struktur rechts in Abbildung 1 ist die Verbindung zwischen w und y aufgeschnitten und ein *Exit/Entry*-Paar eingesetzt. Diese Struktur lässt sich linearisieren und damit umkehrbar eindeutig durch den AA-Ausdruck unten in der Abbildung beschreiben.

4 Physikalische Strukturen

Die Strukturelemente der AA haben bisher nur eine bescheidene Semantik, die sich auf einige allgemeine funktionale und räumliche Eigenschaften bezieht. Um physikalische Systeme vollständig beschreiben zu können, muss man aber noch weitere Eigenschaften definieren. Ein wesentlicher Unterschied besteht z.B. zwischen multiplanaren und echten räumlichen Systemen. Obgleich in dieser Arbeit die Beschreibung echter räumlicher Systeme im Vordergrund steht, erscheint es sinnvoll, multiplanare Systeme nicht ganz unbeachtet zu lassen. Dies vor allem angesichts der enormen Bedeutung, die sie in der Schaltkreistechnik haben.

In der multiplanaren Schaltkreistechnik wird *Next* als eine flexible Kopplung zwischen den verknüpften Teilstrukturen interpretiert, die nur Platz beansprucht, wenn beide Teilstrukturen nicht die gleiche Richtung haben. Für jede andere Richtung gilt der minimale Platz. *Juxta* wird als vertikale Ausrichtung sich berührender Teilstrukturen interpretiert. Ein passendes *Down/Up*-Paar wird als zwei Mengen von Durchkontaktierungen angenommen, die auf einer zweiten Ebene verbunden sind. Ein passendes *Exit/Entry*-Paar wird als Verbindung ohne Platzbedarf interpretiert, ein nichtpassendes *Exit/Entry*-Paar dagegen als die beiden Seiten eines Steckers. *Links* sind echte Leitungen, die in geeigneter Länge überall dort eingesetzt werden, wo unmittelbar abhängige Aktonen nicht aneinander grenzen.

Im Fall echter räumlicher Strukturen werden *Next* und *Juxta* als physikalisch/chemische Bindungen interpretiert. *Down/Up*-Paare und *Exit/Entry*-Paare werden jeweils als Objekte angesehen, die sich auf Grund physikalisch/chemischer Kräfte gegenseitig anziehen. Passende Paare bilden auf diese Weise Verbindungen innerhalb des betrachteten Systems, während einzelne *Down-, Up-, Exit-* oder *Entry*-Aktonen als Andockstellen zu anderen Systemen dienen. Die Kräfte mögen sich in der Reihenfolge *Next*, *Juxta*, *Exit/Entr* , *Down/Up* abschwächen.

Mit diesen Interpretationen beschreibt AA ein lineares physikalisches System, das die Fähigkeit hat, sich selbst in wohldefinierter Weise in ein räumliches physikalisches System zu verwandeln, genauso, wie eine Kette von Aminosäuren sich in ein Protein verwandelt.

Es gibt zwei Basisstrukturen, die aus der Sicht eines Beobachters in nahezu jedem physikalischen System auftreten, die Kreuzung und die topologische Schleife. Eine Kreuzung besteht aus zwei unabhängigen Teilstrukturen, die sich überlagern. Eine Kreuzung ist damit eine dreidimensionale Struktur, in der eine Teilstruktur von der anderen überdeckt wird. Abhängig davon, welche der Teilstrukturen die andere überdeckt, gibt es zwei komplementäre Kreuzungen. In AA lässt sich mittels des *Down/Up*-Paars eindeutig beschreiben, welche der beiden Teilstrukturen unterhalb der anderen liegt. AA hat damit die Fähigkeit, die Chiralität, d.h. den Drehsinn, von Strukturen zu beschreiben.

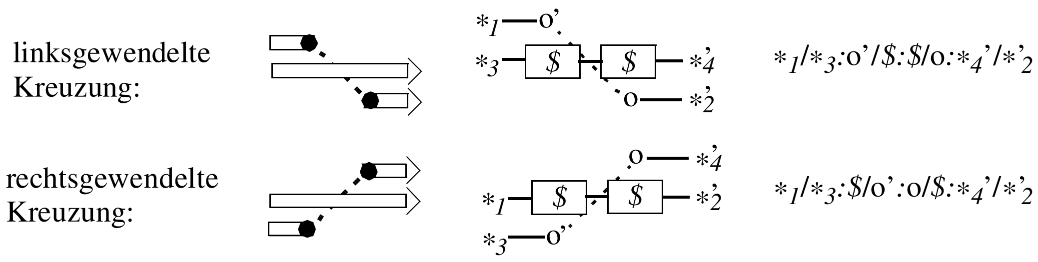


Abbildung 2. Physikalische Struktur, Aktonstruktur und AA-Ausdruck von komplementären Kreuzungen

Abbildung 2 zeigt oben eine linksgewendelte und darunter eine rechtsgewendelte Kreuzung. Dargestellt sind jeweils die eigentliche Struktur, das Gummibandmodell und der AA-Ausdruck. Die schwarzen Punkte in der eigentlichen Struktur markieren *Down* und *Up* und die gepunktete Linie dazwischen ihre unterführendeVerbindung. In biplanarer Umgebung stellen die schwarzen Punkte Durchkontaktierungen dar, in allgemeiner dreidimensionalen Umgebung physikalische Objekte, die sich gegenseitig anziehen.

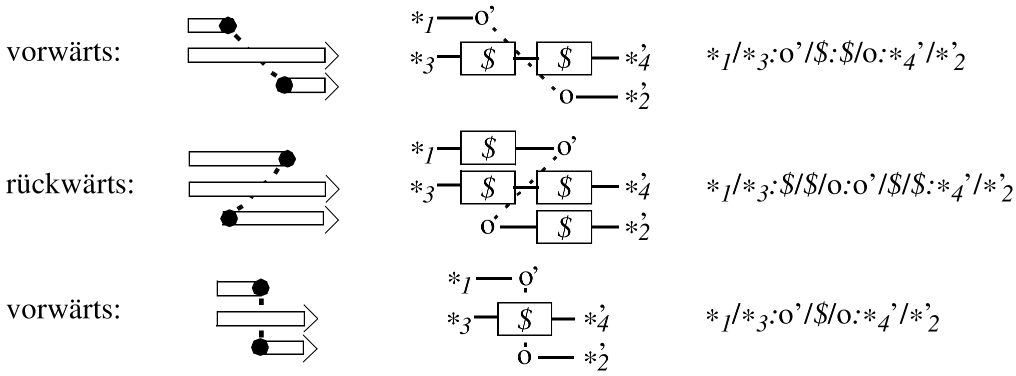
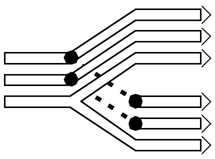


Abbildung 3. Vorwärts- und Rückwärtstypen einer linksgewendelten Kreuzung

Bei genauerer Betrachtung stellen die in Abbildung 2 gezeigten Kreuzungen nur einen Typ dar, der dadurch gekennzeichnet ist, dass das *Down* vor dem *Up* liegt. Entsprechend soll er Vorwärtstyp genannt werden. Dazu gibt es aber auch einen Rückwärtstyp, bei dem *Up* vor *Down* liegt. Der dritte Kreuzungstyp, bei de *Down* und *Up* gleichauf liegen, wird dem Vorwärtstyp zugeschlagen. Abbildung 3 zeigt die drei Typen einer linksgewendelten Kreuzung, wieder dargestellt als eigentliche Struktur, Gummibandmodell und AA-Ausdruck.

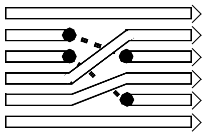
Die Vorwärtskreuzung ist ein essenzieller Bestandteil von zwei Grundstrukturen, die sowohl in biplanaren als auch in allgemeinen räumlichen Systemen auftreten. Sie sollen Fork und Shuffle benannt werden. Fork gabelt ein geordnetes Bündel von Verbindungen in zwei Bündel, die die gleiche Ordnung haben. Shuffle vertauscht die Verbindungen von zwei geordneten Bündeln gleicher Kardinalität so, dass die Ordnung auf Verbindungspaare übertragen wird. Abbildung 4 zeigt beide (linksgewendelten) Grundstrukturen in physikalischer Darstellung und als AA-Ausdruck. Ebenso lassen sich natürlich auch rechtsgewendelte Grundstrukturen beschreiben.

Fork



$*_1/*_2/*_3:$
$(\$/o')/(\$/o')/(\$/\$):(\$/\$/\$)/(o/o/\$):$
$(*'_4/*'_5/*'_6)/(*'_7/*'_8/*'_9)$

Shuffle (S^+):

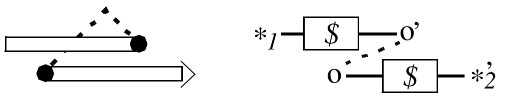


$*_1/*_2/*_3/*_4/*_5/*_6:$
$(\$/o'/o')/(\$/\$/\$):(\$/\$)/(o/\$)/(o/\$):$
$*'_7/*'_8/*'_9/*'_{10}/*'_{11}/*'_{12}$

Abbildung 4. Fork und Shuffle, zwei essenzielle Grundstrukturen

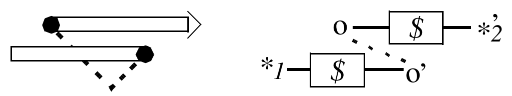
Fork tritt im Allgemeinen nicht explizit in Erscheinung, da es integraler Bestandteil von *Juxta* ist. Fork kann jedoch explizit gemacht werden, wenn das erforderlich ist. Shuffle dagegen ist ein eigenständiges Akton, das dementsprechend für die AA-Beschreibung eine eindeutige Bezeichnung braucht. Es soll mit dem Symbol S^+ bezeichnet werden, wobei das '+' bedeutet, dass es so viele Linienpaare wie notwendig erzeugt. In der Datenverarbeitung tritt ein Shuffle gewöhnlich mit einer nachfolgenden Menge von *And*- oder *Or*-Gattern auf, die jeweils ein Verbindungspaar zu einer Verbindung reduzieren. Diese multiplen Kombinationen können entsprechend zu SA^+ und SO^+ zusammengefasst werden.

linksgewendelte
offene Schleife:



$*_1:\$/o:o'/\$:*'_2$

rechtsgewendelte
offene Schleife:



$*_1:o/\$:\$/o':*'_2$

Abbildung 5. Physikalische Struktur, AA-Struktur und -Ausdruck komplementärer offener Schleifen

Eine topologische Schleife ist nur strukturell. Eine Rückkopplung dagegen ist strukturell und funktionell. Rückkopplungen werden im nächsten Abschnitt behandelt. Eine topologische Schleife kann offen oder geschlossen sein. Eine offene Schleife hat eine enge Beziehung zu Kreuzungsstrukturen. Während jedoch zwei sich überlagernde unabhängige Teilstrukturen sich i.Allg. vorwärts überkreuzen, überkreuzt eine offene Schleife sich selbst immer rückwärts. Eine offene Schleife hat wie eine Kreuzung eine Chiralität. In Abbildung 5 sind eine links- und eine rechtsgewendelte offene Schleife dargestellt, zusammen mit ihrer AA-Struktur und ihrem AA-Ausdruck. Das unterführende Verbindungsband wird wie bisher schon durch eine gepunktete Linie dargestellt. Seine geknickte Form dient lediglich dem Zweck, die Unterführung deutlicher zu zeigen.

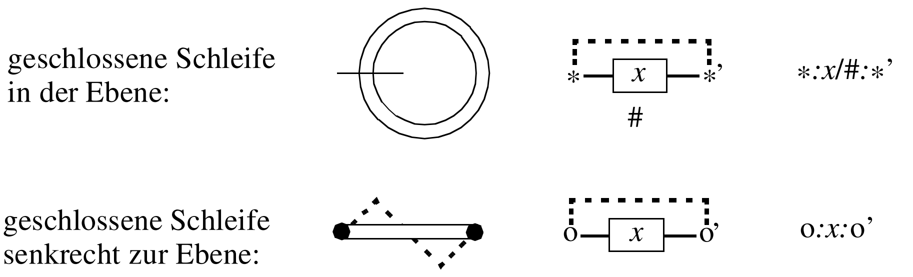


Abbildung 6. Zwei Sichten einer geschlossenen Schleife

Geschlossene Schleifen können sowohl mittels *Exit/Entry*-Paaren als auch *Down/ Up*-Paaren beschrieben werden. Im ersten Fall liegt die geschlossene Schleife in der Beobachtungsebene, im zweiten Fall senkrecht dazu. Beide Fälle sind in Abbildung 6 dargestellt. Bei der AA-Beschreibung im ersten Fall ist allerdings eine Besonderheit zu beachten. Der einfache AA-Ausdruck $*$:*x*:$*$' zeigt nicht, ob der Ring in der Beobachtungsebene links herum oder rechts herum zu schliessen ist, d.h. keinen Drehsinn. Dies Manko lässt sich aber durch Einführung eines *Gap* beheben, da dadurch der planare Abstand zwischen dem *Entry* und dem *Exit* unterschiedlich lang wird. Der zweite Fall hat diese Besonderheit nicht.

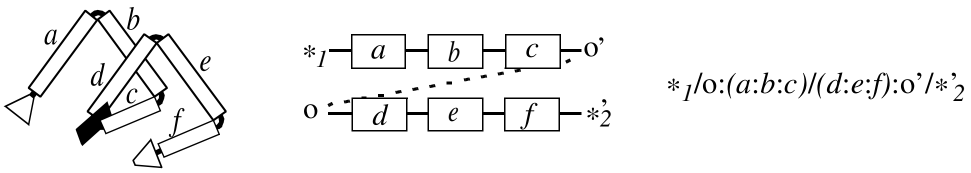
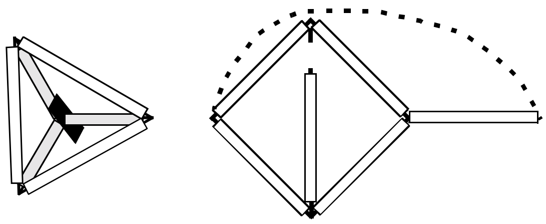


Abbildung 7. Räumliche Struktur, AA-Darstellung und -Beschreibung einer Helix

Wir demonstrieren im Folgenden die Beschreibung offener und geschlossener dreidimensionaler Strukturen durch je ein Beispiel. Das erste Beispiel ist eine Helix, d.h. eine Struktur, die von der Natur sowohl in DNA als auch in Proteinen verwendet wird. Die in Abbildung 7 gezeigte Helix ist recht primitiv, enthält aber sonst keine Beschränkungen. Sie besteht aus zwei Schleifen mit je drei Aktonen, *a, b, c* in der ersten Schleife und *d, e, f* in der zweiten. Für die Beschreibung werden die beiden Schleifen durch Einführung eines *Down/Up*-Paares formal voneinander getrennt. In Abbildung 7 links ist die Trennstelle durch eine kleine schwarze Fläche angedeutet. Diese Trennung erlaubt die Ausbreitung der Helix in der Ebene, wie in der AA-Darstellung in der Mitte gezeigt. Diese wiederum wird umkehrbar eindeutig durch den AA-Ausdruck rechts beschrieben. Hervorzuheben ist, dass die Aktonenbezeichnungen lediglich zur Typunterscheidung dienen. Grundsätzlich ist jedes Akton bereits eindeutig durch seine relative räumliche Position spezifiziert. Im Beispiel wurden die Aktonbezeichnungen lediglich eingeführt, um die Beziehungen zwischen der physikalischen Struktur, der

AA-Struktur und dem AA-Ausdruck zu verdeutlichen.



o:*(e:e/∗)/(e:(e:∗’)/e):e:*o’

Abbildung 8. Räumliche Struktur, planare Darstellung und AA-Ausdruck eines Tetraeders

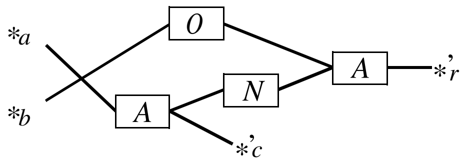
Als Beispiel für eine dreidimensionale geschlossene Struktur soll ein Tetraeder behandelt werden, wie in Abbildung 8 dargestellt. Dazu sei angenommen, dass er gleichlange Kanten hat, die deshalb einheitlich die Typbezeichnug *e* erhalten sollen. Der erste Schritt zur AA-Beschreibung ist die Ausbreitung des Tetraeders auf einer Ebene. Dies lässt sich durch Auftrennung des Tetraeders und Einsetzung eine *Down/Up*-Paars auf seiner Rückseite erreichen, z.B. an der Stelle, die in Abbildung 8 links durch eine kleine schwarze Fläche gekennzeichnet ist. Die sich ergebende planare Struktur enthält noch nicht verschachtelbar. Dies wird erst durch einen weiteren Schnitt und Einsetzung eine *Exit/Entry*-Paars erreicht. Das Ergebnis der beiden Schnitte ist die in der Mitte von Abbildung 8 gezeigte planare Darstellung, die umkehrbar eindeutig durch den AA-Ausdruck links beschrieben wird. Hervorzuheben ist, dass der AA-Ausdruck sogar festlegt, dass bei einer Rekonstruktion des Tetraeders *Down* und *Up* nach hinten zusammengeführt werden. Dies beinhaltet, dass AA auch beschreibt, was an einer Struktur innen und aussen ist.

5 Datenverarbeitung in Proteinen

Im vorigen Abschnitt wurde gezeigt, wie sich unter Verwendung von *Down/Up*- und *Exit/Entry*-Paaren verschiedene räumliche Strukturen beschreiben lassen, darunter die Helix, die einerseits die Basisstruktur von DNA und RNA ist, aber auch in Proteinen eine besondere Rolle spielt. DNA und RNA bestehen aus Doppelhelices, die zwischen sich die Erbinformation tragen. Diese Doppelhelices können ohne Schwierigkeiten mit der AA beschrieben werden. Während aber DNA und RNA nach heutigem Kenntnisstand nur zur Speicherung von Erbinformation dienen und damit passiv sind, führen Proteine Funktionen aus, die eine Datenverarbeitung beinhalten. Wie diese Datenverarbeitung geschieht, ist heute noch weitgehend unklar. Sicher ist nur, dass sie in Form chemisch/physikalischer Effekte in und zwischen Molekülen erfolgt und damit weitaus komplexer ist als die auf elektrischen Effekten beruhende in Computern. Nichtsdestoweniger gibt es keinen Grund anzunehmen, dass die biologische Datenverarbeitung in anderer Weise geschieht als die klassische.

Eine vollständige Beschreibung von Proteinen muss nicht nur die Strukturen, sondern auch die Datenverarbeitung und -speicherung abdecken. AA bietet auch diese Eigenschaften, wie in diesem Abschnitt gezeigt werden soll.

Generell lassen sich in der Datenverarbeitung zwei gerichtete Strukturen unterscheiden, lineare (kombinatorische) Netze und Rückkopplungsnetze, die in der Schaltungstechnik als Schaltnetze und Schaltwerke bezeichnet werden. Schaltnetze können mit den Mitteln der Schaltalgebra stückweise durch Auflistung Boolescher Gleichungen beschrieben werden, für Schaltwerke gibt es bisher keine formale Sprache. Mit der AA dagegen können sowohl lineare als auch Rückkopplungsnetze vollständig beschrieben werden. Dies soll an zwei Beispielen demonstriert werden.

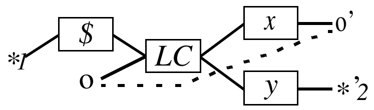


$(*_a/*_b){:}O/(A{:}N/*'_c\,){:}A{:}*'_r$

Abbildung 9. AA-Struktur und -Ausdruck eines Halbaddierers

Als Beispiel für ein lineares Netz wählen wir einen Halbaddierer. Seine AA-Struktur und sein AA-Ausdruck sind in Abbildung 9 gezeigt. Die Entries $*_a/*_b$ liefern den Input, der *Exit* $*'_c$ den Übertrag und *Exit* $*'_r$ das Ergebnis. Die Buchstaben *A*, *O*, *N* bezeichnen ein *And*-Gatter, ein *Or*-Gatter und einen Inverter. Die gezeigte AA-Struktur ist topologisch identisch mit der physikalischen Struktur. Die Funktionen und die Struktur des Halbaddierers werden durch den AA-Ausdruck rechts in der Abbildung präzise beschrieben.

Zum Vergleich sei die schaltalgebraische Beschreibung erwähnt. Sie erfordert zwei Boolesche Gleichungen $a \wedge b = c$ und $(a \vee b) \wedge \neg(a \wedge b) = r$ und sagt zudem nichts über die Schaltungsstruktur aus.

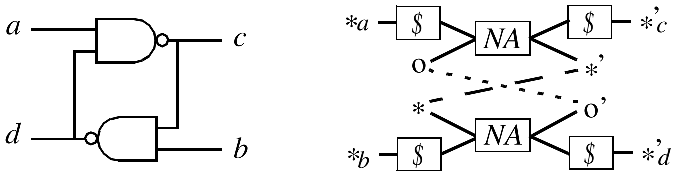


$*_1{:}\$/\mathrm{o}{:}LC{:}(x{:}\mathrm{o}')/y{:}*'_2$

Abbildung 10. Eine (linksgewendelte) Rückkopplungsschleife, kontrolliert durch das Akton *LC*

Als zweites Beispiel betrachten wir eine allgemeine (linksgewendelte) Rückkopplungsschleife, wie in Abbildung 10 gezeigt. Eine Rückkopplungsschleife ist eine aktive offene Schleife, deren Input und Output gemeinsam kontrolliert bzw. verarbeitet wird. In der Abbildung ist das Kontrollakton mit *LC* bezeichnet. Ein primitives *LC* besteht nur aus eine *And*- oder *Nand*-Gatter und erzeugt damit entweder eine positive oder negative Rückkopplung. Erstere führt zu einem einzigen stabilen Zustand, letztere zur Oszillation. Eine positive Rückkopplung allein hat keine sinnvolle Anwendung, die Kombination von zwei verschachtelten positiven Rückkopplungen in Form von zwei *Nand*-Gattern dagegen hat zwei stabile Zustände und ist damit als Speicher geeignet, der als RS-Flipflop bezeichnet wird .

Abbildung 11 zeigt ein solches RS-Flipflop. Die AA-Struktur rechts gleicht in ihrer planaren Topologie der üblichen Schaltbilddarstellung, in der die beiden Eingänge links und die beiden Ausgänge rechts angeordnet sind. Bei räumlicher Interpretation erzwingt jedoch die in Abschnitt 4 eingeführte Attraktionssemantik der *Up/Down*- und *Entry/Exit*-Paare ein Umklappen der unteren Teilstruktur gegenüber der oberen, wie links in der Abbildung gezeigt. Dies ist aber nur eine Struktur unter vielen anderen, die ebenfalls präzise beschrieben werden können.



$$(*_a:\$/\mathrm{o}:NA:\$/*':*'_c)/(*_b:*/\$:NA:\mathrm{o}'/\$:*'_d)$$

Abbildung 11. Räumliche Struktur, AA-Struktur und -Ausdruck eines RS-Flipflops

Literatur

1. Jones, G., Sheeran, M.: Circuit Design in Ruby. In: Staunstrup, J. (ed.): Formal Methods of VLSI Design. North Holland (1990) 13-70
2. Kolla, R., Molitor, P., Osthof, H.G.: Einführung in den VLSI-Entwurf. Teubner-Verlag, Stuttgart (1989)
3. Rivas, E., Addy, S.R.: The language of RNA: a formal grammar that includes pseudoknots. Bioinformatics, Vol. 16, No. 4, , (2000) 334-340
4. Searls, D.B.: Formal Language Theory and Bioloical Macromolecules. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 47 (1999)

Towards Type Systems for Dynamic Components (Abstract)

Arnd Poetzsch-Heffter

FernUniversitäat Hagen

In an object-oriented setting, a system is a collection of communicating objects. Current programming techniques do not support mechanisms to structure such collections, e.g. by providing constructs to group objects and to enforce interfaces of such groups. In the following, we call a group of objects that work closely together a *(dynamic) component*. We assume that two components are either disjoint w.r.t. the objects they contain or that one component is part of the other component, i.e. all objects of the former component belong as well to the latter. Consequently, components induce a hierarchical structure on the objects in a system.

The component structure introduces a boundary between objects inside and outside the component. In particular, given a component C, we can distinguish four kinds of objects:

- internal: objects in C that must not be referenced from the outside;
- interface: objects in C that may be referenced from the outside are called interface objects;
- used: objects outside C that are referenced from objects in C are called used objects;
- external: objects not directly referenced from objects in C.

All the communication of a component with the outside objects can be controlled by the interface and used objects. The types of the interface objects describe what kind of services the component provides; as in all typed OO-languages, the types determine which methods can be invoked on interface objects, i.e. from the outside. The types of the used objects explain what kind of services the component needs from the outside. Based on these types, the behavior of a component can be characterized. In addition to the types, the number of interface and used objects and possibly some information about their relationship might be of interest. In general, the number and relationship of interface and used objects can change dynamically.

In this talk, we present an approach for typing the interfaces of dynamic components. The basic idea is to use regular expressions to control the dynamic change of the number and relationship of interface and used objects. The alphabet of such a regular expression R is given by $\{\text{interface}, \text{used}\} \times$ TYPES where TYPES is the set of reference types of the underlying object-oriented program. If $w_1 \ldots w_n$ is a word in LANG(R), then the component may have n interface and used objects where the kind and type of the i-th reference is constrained by the letter w_i. Relationship information is captured by the derivation trees according to R.

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

Integration von Polymorphismus und Subtypen für den π-Kalkül

Dirk Draheim

Institut für Informatik, Freie Universität Berlin draheim@inf.fu-berlin.de

Zusammenfassung In diesem Papier werden π-Kalkül-Typsysteme mit Polymorphismus einerseits [13] und Subtypen andererseits [8] zu einem neuen π-Kalkül mit beschränktem Polymorphismus integriert. Als motivierendes Beispiel wird die natürliche Darstellung einer generischen Compute-Engine im resultierenden Kalkül präsentiert. Technische Beiträge sind zum einen Beweise der notwendigen Eigenschaften des Typsystems, das heißt ein Subjekt-Reduktionstheorem und Theoreme zur Typ-Stärkung und -Schwächung. Außerdem beweisen wir korrektes Laufzeitverhalten wohlgetypter Prozesse.

Keywords: π-Kalkül, σ-Kalkül, Beweistechniken

1 Einleitung

In diesem Papier möchten wir Fortschritte unserer laufenden Arbeit präsentieren, das σ-Kalkül zweiter Stufe in einem geeigneten beschränkt polymorphen π-Kalkül zu interpretieren.

Der π-Kalkül [6] ist eine Prozeßalgebra, die das Konzept der nachrichtenbasierten Kooperation mittels Kommunikation von Kanalnamen formalisiert. Er erlaubt die Beschreibung von Prozeß-Systemen, in denen sich die Kommunikationstopologie dynamisch ändert. Der π-Kalkül eröffnete eine neue Perspektive der Untersuchung von Konzepten objektorientierter Programmiersprachen [15]. Der σ-Kalkül [1] ist als minimale objektbasierte Programmiersprache mit Objekt-Formation, Methoden-Aufruf und Methoden-Update aufzufassen.

Fokus unserer Arbeit ist der Ansatz, den σ-Kalkül im π-Kalkül zu interpretieren. Dieser Ansatz ist für Typsysteme erster Ordnung bereits gründlich ausgearbeitet. Die Motivation liegt im wechselseitig vertieften Verständnis von Objekten und Prozessen einerseits [11], in der Erschließung neuer Beweistechniken andererseits [5]. Vor diesem Hintergrund gilt unser Interesse nun fortgeschrittenen Typsystemen. Die bestehenden Interpretationen sollen konsequent bzgl. Polymorphismus fortgesetzt werden. Die Schwierigkeit ist dabei, daß sich im Kontext von Prozessen existenzquantifizierte Typen als die natürliche Form von Polymorphismus erwiesen haben [12], während Generizität in objektbasierten Sprachen hingegen durch allquantifizierte Typen modelliert wird, in voller Analogie zu parametrischem Polymorphismus funktionaler Sprachen [9].

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

In Kapitel 2 führen wir den Kalkül $\pi_{\mu,\times,+,:<,\exists:<}$ ein, einen π-Kalkül mit rekursiven Typen, Produkten, Verbundtypen, Subtypen und beschränktem existenzquantifizierten Polymorphismus. Es folgen als Beispiel eine generische Compute-Engine und die Formulierung von Eigenschaften des Typsystems in den Kapiteln 3 und 4. Abschließend wird mit Kapitel 5 die Relevanz der Ergebnisse für die Deutung von Objekten in Prozeßalgebra aufgezeigt.

2 Der Kalkül $\pi_{\mu,\times,+,:<,\exists:<}$

Wir folgen [12] und präsentieren den Kalkül als Erweiterung von CCS-ähnlicher Prozeßalgebra um die Möglichkeit, Kanalnamen zu kommunizieren. Wesentliche Beiträge von Prozeßalgebra sind die Formalisierung des Konzepts nachrichtenbasierter Kommunikation einerseits und die Behandlung von Prozessen als Objekte potentiell unendlichen Verhaltens andererseits. Im π-Kalkül findet sich letzterer Aspekt im Replikationsoperator `bang`. Der π-Kalkül ermöglicht zusätzlich die Modellierung von Systemen mit sich dynamisch ändernder Kommunikationstopologie.

Wir wiederholen zunächst das π-Kalkül mit geeignetem Typsystem für Kanaltypen, Produkten und Verbundtypen und führen anschließend Subtypen und Polymorphismus ein.

2.1 Der Kalkül $\pi_{\mu,\times,+}$ als erweitertes CCS

Als Metavariablen für Kanalnamen werden v, w, x verwendet, in den sich anschließenden Beispielen auch andere Kleinbuchstaben. Außer den Kanalnamen gibt es weitere Basiswerte, die später zusammen mit dem Typsystem eingeführt werden. Prozesse stehen für Aktionssequenzen. Mögliche Aktionen sind Eingabe, Ausgabe und stille Transition. Prozesse werden aus inaktivem Prozeß, Abfolge, Nichtdeterminismus, Nebenläufigkeit, Kanalerzeugung und Vergleich, sowie der Replikation gebildet. Der Prozeß *wrong* repräsentiert einen Laufzeitfehler.

$$
\begin{array}{rcll}
P, Q, R & ::= & 0 & \text{inaktiver Prozeß} \\
 & | & \pi.P & \text{Präfix, Abfolge} \\
 & | & P + Q & \text{Summe, Nichtdeterminismus} \\
 & | & P \mid Q & \text{Komposition, Nebenläufigkeit} \\
 & | & (\nu x : T)P & \text{Restriktion, Kanalerzeugung} \\
 & | & [x = y]P & \text{Vergleich} \\
 & | & !P & \text{bang, Replikation} \\
 & | & \mathit{wrong} & \text{Fehler} \\
 & & & \\
\pi & ::= & v(x) & \text{Eingabe} \\
 & | & \overline{v}w & \text{Ausgabe} \\
 & | & \tau & \text{stille Transition}
\end{array}
$$

Die Semantik des Kalküls ist standardmäßig in geeigneter Form als kantenmarkiertes Transitionssystem gegeben.

Als Metavariablen für Typen werden S, T verwendet. Es gibt Werttypen V, Kanaltypen L und den Verhaltenstyp $\Diamond$.

$$\begin{aligned} S, T &::= V \mid L \mid \Diamond \\ V &::= \mathit{unit} \mid \mathbb{B} \mid \mathbb{Q} \mid \mathbb{Z} \mid \mathbb{N} \mid \mathbb{N}_0 \mid \ldots \end{aligned}$$

Das Typsystem wird als Satz von Ableitungsregeln formalisert, basierend auf geeigneten Typumgebungen zur Typisierung freier Namen.

$$\Gamma \ ::= \emptyset \mid \Gamma, x : L \mid \Gamma, x : V$$

$$\text{TV-Name} \qquad \frac{}{\Gamma, x : T \vdash x : T}$$

Der π-Kalkül geht nun aus dem Basiskalkül durch Gleichsetzung von Wert- und Kanaltypen hervor.

$$V \ ::= \ L$$

Das Typsystem des π-Kalküls definiert die Wohlgetyptheit von Prozessen konstruktionsarm: Prozesse können ausschließlich den Verhaltenstyp besitzen.

$$\text{T-Par} \qquad \frac{\Gamma \vdash P : \Diamond \quad \Gamma \vdash Q : \Diamond}{\Gamma \vdash P | Q : \Diamond}$$

$$\text{T-Sum} \qquad \frac{\Gamma \vdash P : \Diamond \quad \Gamma \vdash Q : \Diamond}{\Gamma \vdash P + Q : \Diamond}$$

$$\text{T-Nil} \qquad \frac{}{\Gamma \vdash 0 : \Diamond}$$

$$\text{T-Rep} \qquad \frac{\Gamma \vdash P : \Diamond}{\Gamma \vdash !P : \Diamond}$$

$$\text{T-Tau} \qquad \frac{\Gamma \vdash P : \Diamond}{\Gamma \vdash \tau.P : \Diamond}$$

$$\text{T-Match} \qquad \frac{\Gamma \vdash v : \#T \quad \Gamma \vdash w : \#T \quad \Gamma \vdash P : \Diamond}{\Gamma \vdash [v = w]P : \Diamond}$$

$$\text{T-Res} \qquad \frac{\Gamma, x : L \vdash P : \Diamond}{\Gamma \vdash (\nu x : L)P : \Diamond}$$

$$\text{T-In} \qquad \frac{\Gamma \vdash v : \#T \quad \Gamma, x : T \vdash P : \Diamond}{\Gamma \vdash v(x).P : \Diamond}$$

$$\text{T-Out} \qquad \frac{\Gamma \vdash v : \#T \quad \Gamma \vdash w : T \quad \Gamma \vdash P : \Diamond}{\Gamma \vdash \overline{v}w.P : \Diamond}$$

Für die eingeführten Basisdatentypen werden geeignete Basiswerte angenommen.

$$basval ::= \star \mid true \mid false \mid \ \ldots -1, 0, 1 \ldots \ \mid \ \ldots -\frac{1}{2} \ldots \frac{1}{2} \ldots \mid \ldots$$

$$\text{TV-Base} \qquad \frac{basval \in B}{\Gamma \vdash basval : B}$$

Für Produkte und variante Verbunde wird lediglich die Grammatik präsentiert. Die Typen sind jeweils kovariant in ihren Komponenten. Für variante Verbunde gilt außerdem die bekannte Subtypstruktur entlang längerer Typen.

$$\begin{array}{rll}
v ::= & < v_1, \ldots, v_n > & \text{Produktkonstruktion} \\
P ::= & with(x_1, \ldots, x_n) = v \ do \ P & \text{Produktdestruktion} \\
T ::= & T_1 \times \ldots \times T_n & \text{Produkttypen} \\
 & & \\
 & l, h & \text{variante Marken} \\
v ::= & l_v & \text{variante Werte} \\
P ::= & case \ v \ of [l_1_(x_1) \rightarrow P_1; \ldots; l_n_(x_n) \rightarrow P_n] & \text{pattern matching} \\
T ::= & [l_1_T_1, \ldots, l_n_T_n] & \text{variante Typen}
\end{array}$$

Für rekursive Typen wird alternativ zur syntaktischen Einführung von `fold` und `unfold` eine geeignete strukturelle Gleichheit $\sim_{type}$ definiert.

$$\begin{array}{rcl}
V & ::= & X \mid \mu X.V \\
L & ::= & \mu X.L
\end{array}$$

$$\text{EQ-Unfold} \qquad \frac{}{\mu X.T \sim_{type} T\{\mu X.T/X\}}$$

$$\text{T-EQ} \qquad \frac{\Gamma \vdash v \ : \ s \quad S \sim_{type} T}{\Gamma \vdash v \ : \ T}$$

2.2 Subtypen

Die Subtypstruktur des π-Kalküls basiert auf der Unterscheidung zwischen Eingabe- und Ausgabekanälen.

$$\begin{array}{rll}
L ::= & iV & \text{Typ für Eingabekanäle} \\
\mid & oV & \text{Typ für Ausgabekanäle}
\end{array}$$

Die Subtypbeziehung wird in Vorgriff auf die Einführung von Polymorphie in Abhängigkeit von Typumgebungen definiert.

Reflexivität $$\frac{}{\Gamma \vdash T \leq T}$$

Transitivität $$\frac{\Gamma \vdash S \leq S' \quad \Gamma \vdash S' \leq T}{\Gamma \vdash S \leq T}$$

SUB-#In $$\frac{}{\Gamma \vdash \#T \leq iT}$$

SUB-#Out $$\frac{}{\Gamma \vdash \#T \leq oT}$$

SUB-In $$\frac{\Gamma \vdash S \leq T}{\Gamma \vdash iS \leq iT}$$

SUB-Out $$\frac{\Gamma \vdash T \leq S}{\Gamma \vdash oS \leq oT}$$

SUB-Both $$\frac{\Gamma \vdash T \leq S \quad \Gamma \vdash S \leq T}{\Gamma \vdash \#S \leq \#T}$$

Die Regeln `T-In` und `T-Out` ersetzen gemeinschaftlich die bisherige Regel `T-In`.

T-In $$\frac{\Gamma \vdash a : iS \quad \Gamma, x : S \vdash P : \Diamond}{\Gamma \vdash a(x).P : \Diamond}$$

T-Out $$\frac{\Gamma \vdash a : oS \quad \Gamma \vdash w : S \quad \Gamma \vdash P : \Diamond}{\Gamma \vdash \overline{a}(w).P : \Diamond}$$

T-Subsumption $$\frac{\Gamma \vdash v : S \quad S \leq T}{\Gamma \vdash v : T}$$

Der Ausgabekanaltyp ist kontravariant in seinem einzigen Argument, der Eingabekanaltyp ist kovariant. Zusammen mit der Regel `ST-Subsumption` ergibt sich, daß über einen Ausgabekanal Werte eines beliebigen Subtyps gesendet werden können. Da ein Subtyp immer ein speziellerer Typ in dem Sinn ist, daß seine Objekte mehr Möglichkeiten besitzen bzw. mehr Bedingungen erfüllen, entspricht dies der Intuition, daß der Typ eines Ausgabekanals für einen Prozeß eine Verpflichtung darstellt. Analog ergibt sich für Eingabekanäle, daß empfangene Werte an Namen beliebigen Obertyps gebunden werden können. Dies entspricht der Vorstellung, daß der Typ eines Eingabekanals für einen Prozeß eine Garantie darstellt. Eine weitere Rechtfertigung der genannten Typvarianzen ergibt sich bei der Deutung eines Paares aus Eingabe- und Ausgabekanal als Funktionsaufruf und der vollen Analogie der Varianzeigenschaften des entsprechenden Funktionstyps zu denen der entstehenden Prozeßkontextlücke.

Für die Basisdatentypen ist ebenfalls eine geeignete Subtypstruktur definiert. Es folgen beispielhafte Ableitungsregeln für die Zahltypen.

$$\frac{}{\Gamma \vdash \mathbb{N} \leq \mathbb{Q}}$$

$$\frac{}{\Gamma \vdash \mathbb{N}_0 \leq \mathbb{N}}$$

2.3 Polymorphismus

Ein existenzquantifizierter Typ ist die unendliche Summe von Objekten, die unter Verwendung eines Repräsentationstypen für einen ansonsten opaken Typen als Implementierung eines Schnittstellentypen gültig sind. Pakete aus Repräsentationstypen und Implementierungen bieten sich deswegen im Kontext von Programmiersprachen zur Modellierung von abstrakten Datenobjekten an. Insgesamt wird somit in [3] und [7] Mechansimen für lose abstrakte Datentypen Bedeutung gegeben. Im Kontext von Prozeßalgebra dienen existenzquantifizierte Typen hingegen natürlicherweise zur Formulierung von Prozessen, die homogen in Abhängigkeit von Typen Dienste erbringen und ermöglichen somit eine Form von parametrischem Polymorphismus. Zu diesem Zweck wird der Kalkül zunächst um das Konzept der beschränkten Typvariablen erweitert.

$$\Gamma ::= \Gamma, X \leq Y \quad \Leftarrow X \notin dom\Gamma$$

$$\text{SUB-Var} \qquad \frac{}{\Gamma, X \leq Y, \Gamma' \vdash X \leq Y}$$

Werte eines existenzquantifizierten Typen sind Pakete aus Typ und Kanal. Die Pakete werden mit `open` geöffnet. Zur weiteren Destruktion im Körper des `open`-Konstrukts wird eine geeignete `with`-pattern-matching-Syntax angenommen, die in den Beispielen im nächsten Kapitel benötigt wird.

$$\begin{array}{rll} V & ::= \exists X \leq S.T & \text{polymorphe Typen} \\ v & ::= [T; v] & \text{Konstruktion} \\ P & ::= open\ v\ as(X \leq T; x) in P & \text{Destruktion} \end{array}$$

Typsystem und Reduktionssemantik werden wie folgt erweitert.

$$\text{T-Open} \qquad \frac{\Gamma \vdash v : \exists X \leq S.T \quad \Gamma, X \leq S, x{:}T \vdash P \;:\; \Diamond}{\Gamma \vdash open\ v\ (X \leq S; x)\ in\ P : \Diamond}$$

$$\text{T-Poly} \qquad \frac{\Gamma \vdash v \;:\; V(W/X)\Gamma \vdash W \leq S}{\Gamma \vdash [W; v] : \exists X \leq S.V}$$

$$\text{POLY} \qquad \frac{}{open[V; w]as(X \leq S, y)\ in\ P \xrightarrow{\tau} P\{V/X\}\{w/y\}}$$

$$\text{POLY-ERR} \qquad \frac{v\ does\ not\ have\ the\ form\ < V; w >}{open\ v\ as\ (X \leq S, y)\ in\ P \xrightarrow{\tau} wrong}$$

3 Eine generische Compute-Engine

Als Beispiel für die Ausdrücksmächtigkeit des resultierenden Kalküls modellieren wir die Berechnungen einer Compute-Engine. In [16] wird der Begriff der generischen Compute-Engine für einen Server geprägt, der Daten und Methoden als Objekt gekapselt erhält, um diese lokal zur Ausführung zu bringen. In diesem Szenario wird der objektorientierte Subtypmechanismus genutzt, um Minimalanforderungen an die übermittelten Tasks zu spezifizieren. Zur Einführung betrachten wir die Implementierung der polymorphen Identitätsfunktion.

$$ID \stackrel{DEF}{\equiv} a(t).open\ t\ as(X \leq \mathbb{Q})\ in\ (with(x,y) = t\ do\ \overline{x}y).0$$

Es folgt die Anwendung der Identitätsfunktion auf zwei Pakete, die Daten unterschiedlichen Typs transportieren.

$$\begin{array}{ll} & a : \#(\exists X \leq \mathbb{Q}.(X \times oX)\ ,\ b : o\mathbb{N}\ ,\ c : o\mathbb{Q} \\ \vdash & \\ & (\overline{a}[\mathbb{N}; < 3, b >]) \mid \overline{a}[\mathbb{Q}; < \frac{3}{4}, c >] \mid !ID) \\ \stackrel{\tau}{\rightarrow}^{*} & \\ & (\overline{b}3.0 \mid \overline{c}\frac{3}{4}.0 \mid !ID) \end{array}$$

Um in unserem Beispiel bessere Analogie zu dem Beispiel in [16] herstellen zu können bedarf es eigentlich eines π-Kalküls höherer Ordnung [10]. Wir begnügen uns aber mit Betrachtung der Verantwortlichkeit der Compute-Engine, unter der die als Dienst zu erbringende Berechnung durchgeführt wird.

$$CE \stackrel{DEF}{\equiv} !a(t).open\ t\ as(X \leq \mathbb{Q})\ in\ (with(v,w,x,y) = t\ do\ \overline{w}v.x(z).\overline{y}z)$$

wobei
$v \mathrel{\hat{=}}$ Wert
$w \mathrel{\hat{=}}$ Methodenauruf
$x \mathrel{\hat{=}}$ Methodenausführung
$y \mathrel{\hat{=}}$ Endergebnis

Es folgt die Anwendung der Compute-Engine auf zwei Tasks verschieden getypten Inhalts.

$$\begin{array}{ll} & a : \#[\exists X \leq \mathbb{Q}.X \times oX \times iX \times oX] \\ \vdash & \\ & (\nu b)(b(x).0 \mid (\nu m)(\nu n)(\overline{a}[\mathbb{N}; < 3, b, m, n >].m(x).\overline{n}x.0)) \\ & \mid (\nu c)(c(x).0 \mid (\nu m)(\nu n)(\overline{a}[\mathbb{N}; < \frac{3}{4}, c, m, n >].m(x).\overline{n}x.0)) \\ & \mid !CE \\ \stackrel{\tau}{\rightarrow}^{*} & \\ & (\nu b)(b(x).0 \mid \overline{b}3.0)) \mid (\nu c)(c(x).0 | \overline{c}\frac{3}{4}.0) \mid !CE) \\ \stackrel{\tau}{\rightarrow}^{*} & \\ & !CE \end{array}$$

Der Task besteht in beiden Fällen in Ausführung der Identitätsfunktion. Es ist zu beachten, wie durch die Verwendung von Restriktionen die Ausführung jeweils in die Verantwortlichkeit der Compute-Engine verbracht wird.

4 Theoreme für den $\pi_{\exists<}$-Kalkül

Wir formulieren nun die grundlegenden Eigenschaften des Typsystems. Sie werden in der fortlaufenden Arbeit, die im anschließenden Kapitel beschrieben wird, benötigt.

Lemma 41 (Korrektheit wohlgetypter Terme) *Vorausgesetzt :* $\Gamma \vdash P : \Diamond$. *Dann folgt :* P *enthält* wrong *nicht als Subterm.* $\Box$

Lemma 42 (Schwächung der Umgebung) *Vorausgesetzt :* $\Gamma \vdash P : \Diamond \;\wedge\; x \notin dom\Gamma \;\wedge\; S \notin \Gamma$. *Dann folgt :* $\Gamma, x : S \vdash P : \Diamond$. $\Box$

Lemma 43 (Stärkung der Umgebung) *Vorausgesetzt :* $\Gamma, x : S \vdash P : \Diamond \;\wedge\; x \notin fn(P)$. *Dann folgt :* $\Gamma \vdash P : \Diamond$. $\Box$

Lemma 44 (Typverengung für Werte) *Vorausgesetzt :* $\Gamma, p : S \vdash w : U \;\wedge\; T \leq S$. *Dann folgt :* $\Gamma, p : T \vdash w : U$. $\Box$

Lemma 45 (Typverengung für Prozesse) *Vorausgesetzt :* $\Gamma, p : S \vdash P : \Diamond \;\wedge\; T \leq S$. *Dann folgt :* $\Gamma, p : T \vdash P : \Diamond$. $\Box$

Lemma 46 (Substitution) *Vorausgesetzt :* $\Gamma \vdash P : \Diamond \;\wedge\; \Gamma \vdash x : T \;\wedge\; \Gamma \vdash v : T$. *Dann folgt :* $\Gamma \vdash P\{x/v\} : \Diamond$.

Beweis: Dieses Lemma folgt mittels Analyse der möglichen Ableitungssequenzen aus Lemma 42 . $\Box$

Theorem 1 (Subjekt-Reduktion) *Vorausgesetzt :*

$$\Gamma \vdash P : \Diamond \;\wedge\; P \xrightarrow{\alpha}_E P'$$

Dann folgt :

1. $\alpha = \tau \Rightarrow \Gamma \vdash P' : \Diamond$
2. $\alpha = av \Rightarrow \exists T.$
   (a) $\Gamma \vdash a : iT$
   (b) $\Gamma \vdash vT \Rightarrow \Gamma \vdash P' : \Diamond$
3. $\alpha = (\nu \tilde{x} : \tilde{S})\overline{a}v \Rightarrow \exists T.$
   (a) $\Gamma \vdash a : oT$
   (b) $\Gamma, \tilde{x} : \tilde{S} \vdash v : T$
   (c) $\Gamma, \tilde{x} : \tilde{S} \vdash P' : \Diamond$
   (d) *jede Komponente von* $\tilde{S}$ *ist ein Kanaltyp*

Beweis: Der Beweis erfolgt durch strukturelle Induktion über die Länge der Mehrschritt-Transition bzgl. der gewählten operationellen Semantik mittels Lemma 42 bis Lemma 46. $\Box$

Korollar 1 (Laufzeit-Korrektheit) *Vorausgesetzt :* $\Gamma \vdash P : \Diamond \;\wedge\; P \xrightarrow{\alpha} P'$. *Dann folgt :* P' *enthält* wrong *nicht als Subterm.*

Beweis: Das Korollar folgt aus Lemma 41 und Theorem 1. $\Box$

5 Ausblick

Der σ-Kalkül [1] ist eine minimale objektbasierte Programmiersprache mit Objekt-Formation, Methoden-Aufruf und Methoden-Update. Er hat wesentlich zum Verständnis zentraler Konzepte objektorientierter Sprachen und deren Typsysteme, wie z.B. Selbstreferenz und kovariante Selbst-Typen, beigetragen.

Ziel der fortlaufenden Arbeit ist es, den σ-Kalkül zweiter Stufe im π-Kalkül zu interpretieren. Für den σ-Kalkül erster Stufe ist die Interpretation bereits erfolgreich ausgearbeitet worden. Sie benötigt als Zielsprache $\pi_{\mu,+,:<}$, den π-Kalkül mit rekursiven Typen, varianten Verbunden und Subtypen.

Die existierende Interpretation ist allerdings nicht vollständig abstrakt. Die laufende Arbeit versucht deshalb konsequent, Ergebnisse aus dem Bereich der Deutung von Objekten mittels prozeduraler Konzepte nutzbar zu machen [4] [2] [14].

Literatur

1. Martin Abadi, Luca Cardelli. A Theory of Objects. Monographs in Computer Science, Springer-Verlag, 1996
2. Martin Abadi, Luca Cardelli. An Interpretation of Objects and Object Types. Proceedings of POPL '96, ACM, 1996
3. Luca Cardelli, Peter Wegener. On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, Vol. 17, No. 4, December 1985
4. Samuel Kamin. Inheritance in smalltalk-80: A denotational definition. In: 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 80–87, January 1988.
5. Josva Kleist. Reasoning about Objects using Process Techniques, Ph.D. Thesis. Aalborg University, 2000
6. Robin Milner, Joachim Parrow, David Walker. A Calculus for Mobile Processes, Part I/II. Information and Computation 100, pp. 1-40/41-77, 1992
7. John C. Mitchell, Gordon D. Plotkin. Abstract Types have Existential Type. ACM Transaction on Programming Languages and Systems,vol. 10, no. 3, 1988
8. Benjamin Pierce, Davide Sangiorgi. Typing and Subtyping for Mobile Processes. In: Mathematical Structures in Computer Science, pp. 409-454, 1996
9. John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In: Information Processing 83, Elsevier Science Publishers, 1983
10. Davide Sangiorgi. From pi-calculus to Higher-order pi-calculus and back. In: TAPSOFT, Springer Verlag, LNCS 668, 1993.
11. Davide Sangiorgi. An Interpretation of Typed Objects into π-Calculus. In: Information and Computation, pp. 34-73, Academic Press, 1998
12. Davide Sangiorgi. David Walker. The π-calculus: a Theory of Mobile Processes. Cambridge University Press. June 2001, to appear
13. David N. Turner. The Polymorphic π-Calculus: Theory and Implementation. Ph.D. Thesis. University of Edinburgh. 1996
14. Ramesh Vismanathan. Full Abstraction for First-Order Objects with Recursive Types and Subtyping. In: Logic of Computer Science, 1998
15. David Walker. π-calculus Semantics of Object-Oriented Programming Languages. In: Theoretical Aspects of Computer Science, pp.532-547. Springer-Verlag, 1991
16. Jim Waldo, Ann Wollrath. RMI Tutorial. Sun Microsystems, 1999

Realisierung rekursiver Datenstrukturen durch generische Klassen

Andreas Vox

Institut für Softwaretechnik und Programmiersprachen
Medizinische Universität zu Lübeck
http://www.isp.mu-luebeck.de
vox@isp.mu-luebeck.de

Zusammenfassung Wir stellen das Generic-Factory-Muster vor, welches der Realisierung rekursiver Datenstrukturen in objektorientierten Sprachen dient. Dieses Muster ist eine Weiterentwicklung des bekannten Composite-Musters und nutzt den in Sprachen wie Pizza [12] oder Generic Java [3] vorhandenen parametrischen Polymorphismus, um Ergebnisse aus der gut erforschten funktionalen Welt in objektorientierten Programmen nutzbar zu machen.

1 Einführung

Rekursive Datenstrukturen beschreiben baumartige Strukturen, in denen Teilbäume den gleichen Typ wie die Wurzel haben. Derartige Strukturen werden von fast allen Programmen benötigt; Beispiele reichen von einfachen Listen bis zu abstrakten Syntaxbäumen.

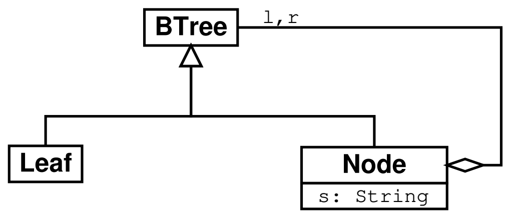
Der Aufbau von rekursiven Datenstrukturen läßt sich auf drei Bildungsprimitiva zurückführen: Tupelbildung (direktes Produkt), Alternativenbildung (direkte Summe) und Rekursion.

In funktionalen Sprachen lassen sich rekursive Datenstrukturen direkt als algebraischer Datentyp definieren:

```
datatype BTree = Leaf | Node of BTree * string * BTree
```

Hier wird Tupelbildung durch * und Alternativenbildung durch | dargestellt. Rekursion wird über Namensgleichheit erreicht.

In der objektorientierten Programmierung werden rekursive Datenstrukturen üblicherweise durch das Composite-Entwurfsmuster [5] realisiert. Danach wird die durch den obigen SML-Datentyp repräsentierte Datenstruktur durch drei Datenklassen implementiert, die folgende Klassenstruktur aufweisen:



W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

Tupelbildung wird durch Aggregation bzw. dem Einfügen von Attributen ausgedrückt. Alternativenbildung entspricht der Spezialisierung durch Unterklassen. Rekursion wird durch Aggregation von Elementen der Oberklasse erreicht.

Bei komplexen Datenstrukturen – wie z.B. abstrakten Syntaxbäumen – wird für jede Alternative eine zusätzliche Datenklasse benötigt wird. Der Entwurf wird dadurch unhandlich und die Sichtweise verschiebt sich von der rekursiven Datenstruktur als Ganzen hin zu den einzelnen Datenklassen.

Unsere Lösung erweitert das Composite-Muster um ein Generic-Factory-Interface, welches die *Struktur* der Datenklassen beschreibt, ohne die *Typen* der Datenklassen festzulegen. Implementierungen dieses Interfaces haben die Aufgabe, konkrete Instanzen der Datenklassen zu erzeugen.

Eine `reduce`-Methode in den Datenklassen erlaubt es, Implementierungen des Generic-Factory-Interfaces ähnlich wie Visitors [5] zu verwenden. Im Gegensatz zum Visitor-Muster ist der Typ der Generic-Factory nicht von den Datenklassen abhängig. Dies ermöglicht es, Generic-Factories zur Beschreibung von Modulschnittstellen zu benutzen, ohne das Modul auf einen konkreten Datentyp festzulegen.

Diese Arbeit gliedert sich wie folgt: In Abschnitt 2 stellen wir das Generic-Factory-Muster vor und illustrieren die Technik durch mehrere Anwendungsbeispiele. Im Abschnitt 3 gehen wir auf den praktischen Einsatz dieses Musters in der Programmentwicklung ein. Der Zusammenhang mit algebraischen Methoden wird kompakt in Abschnitt 4 beschrieben. Abschnitt 5 schließt mit einer Zusammenfassung und einem Ausblick ab.

2 Das Generic-Factory-Muster

Die Hauptidee des Generic-Factory-Musters ist es, die Strukturbeschreibung einer rekursiven Datenstruktur von den Datenklassen zu trennen. Dadurch eröffnet sich die Möglichkeit, die Datenstruktur auf verschiedene Weisen zu implementieren, von denen das Composite-Muster eine „kanonische“ Realisierung beschreibt. In Abschnitt 2.1 stellen wir Generic-Factory-Interfaces vor, welche die typunabhängige Beschreibung von rekursiven Datenstrukturen leisten. In Abschnitt 2.2 beschreiben wir, wie die Datenklassen des Composite-Musters mit dem Generic-Factory-Interface zusammenhängen und wie sie über die `reduce`-Methode mit anderen Realisierungen derselben Datenstruktur zusammenarbeiten.

Darauf aufbauend zeigen wir in Abschnitt 2.3, wie das Generic-Factory-Interface benutzt werden kann, um Schnittstellen zwischen Komponenten zu beschreiben, die Exemplare dieser Datenstruktur produzieren und/oder konsumieren. Interessanterweise hat das zur Folge, dass diese Datenstrukturen gar nicht mehr erzeugt werden müssen.

2.1 Eine abstrakte Beschreibung für rekursive Datenstrukturen

Ein erster Ansatz ist es, die Factory-Klasse zu betrachten, welche die Exemplare der dem Composite-Muster entsprechenden Datenklassen erzeugt:

```
class BTreeFactory {
    BTree caseLeaf() { ... }
    BTree caseNode(BTree l, String s, BTree r) { ... }
}
```

Diese Klasse enthält bereits sämtliche Informationen über die Datenstruktur:

- `BTree`s sind aus den Klassen `Leaf` und `Node` aufgebaut und
- ein `Node` enthält zwei Teilbäume `l` und `r` sowie ein Datum `s` vom Typ `String`.

Der nächste Schritt besteht darin, diese Klasse zu generalisieren, indem die erzeugte Klasse durch einen Typparameter ersetzt wird:

```
interface GenericBTreeFactory<X> {
    X caseLeaf();
    X caseNode(X l, String s, X r);
}
```

Diese Form nennen wir ein Generic-Factory-Interface.

Um zu sehen, wie dieses Interface eine rekursive Datenstruktur beschreibt, kann es als rekursive Typgleichung

$$X = \texttt{unit} \mid X \times \texttt{String} \times X$$

gelesen werden (`unit` sei der einelementige Datentyp $\{\texttt{()}\}$).

(Seiteneffektfreie) Implementierungen von `GenericBTreeFactory` beschreiben Funktionen vom Typ $f : \texttt{unit} \mid X \times \texttt{String} \times X \to X$. Falls f bijektiv ist, so erfüllt X die obige Typgleichung. Dies ist zum Beispiel für `BTreeFactory` der Fall.

Im folgenden Abschnitt gehen wir darauf ein, wie beliebige (nicht bijektive) Implementierungen von `GenericBTreeFactory` sinnvoll mit `BTree`s zusammenarbeiten können.

2.2 Die `reduce`-Methode in den Datenklassen

Bei gegebenem Generic-Factory-Interface lassen sich die konkreten Datenklassen nach dem Composite-Muster automatisch erzeugen. Zusätzlich erhalten alle Datenklassen eine generische Methode

```
<R> R reduce (GenericBTreeFactory<R> fab);
```

welche in den Unterklassen jeweils die entsprechende Methode in `fab` aufruft. Dabei wird `reduce` rekursiv für diejenigen Felder aufgerufen, die vom Typ der Oberklasse sind.

Im Beispiel ergibt das folgenden Code:

```
abstract class BTree {
    abstract <R> R reduce(GenericBTreeFactory<R> fab);
}
class Leaf extends BTree {
    <R> R reduce(GenericBTreeFactory<R> fab) {
        return fab.caseLeaf();
}   }
class Node extends BTree {
    BTree l; String s; BTree r;
    <R> R reduce(GenericBTreeFactory<R> fab) {
        return fab.caseNode(l.reduce(fab), s, r.reduce(fab));
}   }
```

Algorithmisch realisiert **reduce** einen bottom-up Baumdurchlauf, wobei der Generic-Factory-Parameter angibt, wie die Zwischenergebnisse kombiniert werden.

Der wichtige Punkt ist, dass das `GenericBTreeFactory`-Interface ohne Verwendung von `BTree` oder seinen Unterklassen implementiert werden kann. Im Zusammenspiel mit `reduce` lassen sich so Algorithmen unabhängig vom Typ der konkreten Datenklassen formulieren. Beispielsweise liefert die Klasse

```
class Counting implements GenericBTreeFactory<Integer> {
    Integer caseLeaf() {
        return new Integer(1); }
    Integer caseNode(Integer l, String s, Integer r) {
        return new Integer(1 + l.intValue() + r.intValue()); }
}
```

über den Aufruf t.`reduce(new Counting())` die Anzahl der Knoten eines Baumes t . Ein weiteres Beispiel ist die folgende Klasse, welche die Konkatenation aller `String`-Daten in einem `Btree` liefert:

```
class AsString implements GenericBTreeFactory<String> {
    Integer caseLeaf() {
        return ""; }
    Integer caseNode(String l, String s, String r) {
        return l + s + r; }
}
```

Als drittes Beispiel repräsentiert die Klasse `BTreeFactory` die identische Abbildung auf `BTree`: der Aufruf t.`reduce(new BTreeFactory())` liefert eine tiefe Kopie von t .

Allerdings lässt sich nicht jede Abbildung f : `BTree` $\to M$ durch die Implementierung einer Generic-Factory ohne Seiteneffekte realisieren, ein Gegenbeispiel ist etwa die Funktion `insert`: `String` $\times$ `Btree` $\to$ `BTree` . Eine Implementierung von `insert` müsste sich festlegen, ob sie für ein `Leaf()` den Wert `Leaf()` oder den Wert `Node(Leaf(), s, Leaf())` zurückgibt. Im ersten Fall würde der Baum lediglich kopiert, im zweiten Fall würde `s` an allen Blättern eingefügt. Dieses Problem ließe sich durch Seiteneffekte umgehen; die Implementierung als Methode oder als Visitor ist jedoch eleganter.

2.3 Effiziente modulare Algorithmen

Generic-Factories lassen sich auf drei Weisen zur Definition von Schnittstellen zwischen Algorithmen benutzen:

A) Algorithmen, die einen Parameter vom Typ einer Generic-Factory haben, bauen rekursive Datenstrukturen auf; zum Beispiel ein Parser, der einen abstrakten Syntaxbaum erzeugt.

B) Algorithmen, die als Generic-Factory implementiert sind, konsumieren rekursive Datenstrukturen und erzeugen daraus ein Ergebnis. Ein Beispiel ist die Auswertung von synthetisierten Attributen in einem abstrakten Syntaxbaum.

C) Algorithmen, die sowohl als Generic-Factory implementiert sind als auch einen Parameter vom Typ einer (nicht notwendig derselben) Generic-Factory haben, transformieren eine rekursive Datenstruktur in eine andere. Ein Beispiel ist die Umwandlung eines konkreten Syntaxbaums in den entsprechenden abstrakten Syntaxbaum.

Durch Parameter-Instantiierung lassen sich diese Algorithmen zu neuen Algorithmen zusammensetzen. Übergibt man etwa einem Parser vom Typ A einen passenden Attributauswerter vom Typ B, so erhält man einen Parser, der direkt das synthetisierte Attribut liefert und *dabei den Syntaxbaum gar nicht mehr erzeugt*. Dieser Effekt ist aus der funktionalen Programmierung als Shortcut Deforestation bekannt [16][15].

Als Beispiel sei ein Parser und eine einfache Generic-Factory für Listenstrukturen gegeben:

```
interface GenericListFactory<L> {
    L caseNil();
    L caseCons(String head, L tail);
}
class Length implements GenericListFactory<Integer> {
    Integer caseNil() {
        return new Integer(0); }
    Integer caseCons(String head, Integer tail) {
        return new Integer(1 + tail.intValue()); }
}
static <R> R parse(String eingabe, GenericListFactory<R> fab) {
    int pos = s.indexOf(";");
    if (pos < 0) {
        return fab.caseNil(); }
    else {
        return
         fab.caseCons(s.substring(0, pos), parse(s.substring(pos+1)));
}   }
```

Der Aufruf `parse("a;b;c;d;", new Length())` liefert die Länge der durch `"a;b;c;d;"` repräsentierten Liste, ohne diese Liste zu konstruieren.

Als Beispiel für Typ-C-Algorithmen dient folgende Klasse, die eine Listenstruktur in einen entarteten Binärbaum transformiert:

```
class List2BTreeConverter<T> implements GenericListFactory<T> {
    GenericBTreeFactory<T> fab;
    List2BTreeConverter(GenericBTreeFactory<T> fab) {
        this.fab = fab; }
    T caseNil() {
        return fab.caseLeaf();
    }
    T caseCons(String head, T tail) {
        return fab.caseNode(fab.caseLeaf(), head, tail);
    }
}
```

Der Generic-Factory-Parameter wird dem Konstruktor übergeben und als Referenz in `List2BTreeConverter` gespeichert. Über folgende Pipeline ist es jetzt möglich, die für `BTree`s definierte Klasse `AsString` auch auf Listen anzuwenden:

```
GenericBTreeFactory<String> pipe1 = new AsString();
GenericListFactory<String> pipe2 = new List2BTreeConverter(pipe1);
String text = parse(pipe2);
```

Durch die automatische Shortcut Deforestation werden die dazwischenliegenden Datenstrukturen nicht aufgebaut:

$$\boxed{\texttt{parse}} \xrightarrow{(List)} \boxed{\texttt{List2BTreeConverter}} \xrightarrow{(BTree)} \boxed{\texttt{AsString}} \longrightarrow \texttt{String}$$

3 Programmentwicklung mit Generic-Factories

In diesem Abschnitt soll gezeigt werden, wie sich das Generic-Factory-Muster in den Entwicklungsprozess einordnet. Die Verwendung von Generic-Factories gliedert sich in drei aufeinander aufbauende Schritte:

1. Entwurf einer rekursiven Datenstruktur als Generic-Factory-Interface.
   Ein Generic-Factory-Interface beschreibt kompakt die Alternativen und Felder einer rekursiven Datenstruktur. Die Datenstruktur wird dadurch als eine Einheit und nicht als Sammlung kooperierender Klassen aufgefasst, wie im Composite-Muster.
2. Entwurf von Algorithmen und Modulen, die mit rekursiven Datenstrukturen arbeiten.
   Der Einsatz von Generic-Factories zur Definition von Modulschnittstellen bietet den Vorteil, dass nur die *Struktur*, nicht jedoch der *Typ* der Daten festgelegt wird. Da die Struktur für die meisten Algorithmen wichtiger ist als der konkrete Typ, eröffnet dies mehr Möglichkeiten für Kombination und Wiederverwendung.
   Zusätzlich kann das Visitor-Muster in den Fällen eingesetzt werden, in denen eine Implementierung als Generic-Factory nicht möglich oder sinnvoll ist.

3. Automatische Codegenerierung für die Datenklassen.
   Wir entwickeln zur Zeit einen Codegenerator, der aus einem Generic-Factory-Interface die zugehörigen Datenklassen erzeugt. Die verbleibenden Entwurfsentscheidungen für die Implementierung der rekursiven Datenstruktur werden in Vorlagen für den Codegenerator gekapselt, um so eine gleichförmige Implementierung der Datenklassen zu gewährleisten. Manuelle Änderungen an einzelnen Datenklassen sind nicht wünschenswert und auch nicht nötig, da Visitors, Generic-Factories und externe Tabellen ausreichen, um zusätzliche Funktionalität zu implementieren.
   Der eingesetzte Codegenerator sollte außer den Datenklassen auch das passende Visitor-Interface und eine Reihe von Standardalgorithmen (Iteratoren, Depth-First-Traversal, Level-Order-Traversal, equals, Ein-/Ausgabe, etc.) erzeugen.

4 Vergleich zu algebraischen Ansätzen

Die meisten Ideen für das Generic-Factory-Muster stammen aus der Welt der funktionalen Programmiersprachen [2][4], polytypischer Programmierung [10] und der universellen Algebra [11][7]. In Abschnitt 4.1 fassen wir kompakt die algebraischen Grundlagen rekursiver Datentypen zusammen und setzen sie in 4.2 zum Generic-Factory-Musters in Beziehung. Abschnitt 4.3 zeigt die Grenzen dieser Entsprechung auf.

4.1 F-Algebren

Ein (polynomialer) Funktor ist eine durch Konstanten, direkte Summe und direktes Produkt gebildete Mengenabbildung, die gleichzeitig die Funktionenräume aufeinander abbildet. Ist etwa $\phi(M) := \{()\} + A \times M$ ein Funktor, so wird eine Funktion $f : M \to M'$ durch ϕ auf die Funktion $\phi(f) : \phi(M) \to \phi(M'), \phi(f) = \mathrm{id}_{\{()\}} + \mathrm{id}_A \times f$ abgebildet ($\phi(f)(()) = ()$ und $\phi(f)(a,x) = (a, f(x))$).

Eine F-Algebra $(A, a : \varphi(A) \to A)$ ist ein Paar aus einer Trägermenge A und einer Strukturoperation a. Die Strukturoperation legt fest, wie aus Konstanten und Werten aus A neue Werte in A gebildet werden können.

Eine Varietät ist die Klasse aller F-Algebren, die denselben Funktor ϕ benutzen. Ein Homomorphismus ist eine Struktur erhaltende Abbildung zwischen F-Algebren der gleichen Varietät. Eine F-Algebra $(I, i : \phi(I) \to I)$ ist initial, wenn es zu jeder F-Algebra $(A, a : \phi(A) \to A)$ genau einen Homomorphismus $(\!|a|\!)$ gibt, der (I, i) auf (A, a) abbildet. Diese Homomorphismen werden auch Catamorphismen [2] [1] genannt; zu ihnen sind in der funktionalen Welt eine Reihe von Umformungsgesetzen bekannt.

Ein für die Semantik von algebraischen Datentypen wichtiges Ergebnis ist, dass es in allen Varietäten, die auf einem polynomialen Funktor beruhen, eine initiale Algebra gibt, die bis auf Isomorphie eindeutig ist [6]. Anders ausgedrückt definiert jeder polynomiale Funktor eine Datenstruktur.

4.2 Entsprechungen im Generic-Factory-Muster

Im Generic-Factory-Muster beschreibt das Generic-Factory-Interface genau die Signatur der Strukturabbildungen $a : \phi(A) \to A$. Ein Generic-Factory-Interface definiert daher eine Varietät. Jede seiteneffektfreie Implementierung definiert eine Strukturabbildung und legt über den Typparameter die Trägermenge fest, daher sind seiteneffektfreie Implementierungen des Generic-Factory-Interfaces nichts anderes als F-Algebren. Die Generic-Factory, welche die Datenklassen nach dem Composite-Muster erzeugt, entspricht der initialen Algebra, wobei die `reduce`-Methode den Homomorphismus $(\!| \cdot |\!)$ von der initialen Algebra in eine beliebige andere Algebra realisiert.

4.3 Unterschiede

Die oben aufgezählten Entsprechungen sind nur Idealvorstellungen, die in der imperativen objektorientierten Welt nicht erfüllt sein müssen. Es gibt drei offensichtliche Probleme, welche die direkte Übertragung algebraischer Methoden auf objektorientierte Programmiersprachen wie Java behindern.

Der erste Unterschied ist die Möglichkeit von Seiteneffekten und Nichtdeterminismus. In der funktionalen Welt werden Seiteneffekte durch Monaden [17] gekapselt oder durch ein Effektsystem beschrieben [18]. Die Übertragung dieser Methoden auf objektorientierte Programme steht noch aus.

Ein subtilerer Unterschied ist das Vorhandensein einer Objektidentität. Dies impliziert, dass in der objektorientierten Welt eine feinere Gleichheit existiert als in der algebraischen Welt. Ein Ausweg kann sein, statt `==` nur die strukturelle Gleichheit mittels `equals` zuzulassen.

Eine weitere Besonderheit objektorientierter Sprachen ist der Nullwert. Da `null` automatisch Element jedes neu definierten Datentyps ist, werden die als Generic-Factories aufgeschriebenen Funktoren automatisch als $\phi(M) + \{\texttt{null}\}$ interpretiert. Dies ist allerdings keine wesentliche Einschränkung.

5 Zusammenfassung und Ausblick

Das Generic-Factory-Muster stellt den Entwurf rekursiver Datenstrukturen auf eine abstraktere Ebene, da die Struktur allein aus dem Generic-Factory-Interface ersichtlich ist, und die einzelnen Datenklassen für das Verständnis nicht mehr wichtig sind. Die Formulierung von Algorithmen als Generic-Factory oder als Visitor läuft auf der gleichen Abstraktionsebene ab. Die Datenklassen können später durch ein Werkzeug automatisch generiert werden.

Modulschnittstellen, die mit Hilfe eines Generic-Factory-Interface beschrieben werden, legen nur die Struktur der rekursiven Datenstruktur fest, nicht den konkreten Typ. Die Module werden dadurch leichter wiederverwendbar. Als zusätzlicher Vorteil ist die Komposition solcher Module effizienter als der traditionelle Austausch von Datenstrukturen, da zwischen den Modulen automatisch Shortcut Deforestation stattfindet.

Verwandte Arbeiten im objektorientierten Bereich schlagen andere Wege ein als wir: Pizza [12], eine Erweiterung von Java und ein Vorläufer von GJ [3], kennt algebraische Datentypen und switch-Anweisungen über algebraischen Datentypen. Diese Erweiterungen sind aber nicht generisch und erlauben daher nicht die selbe Flexibilität wie unsere Lösung. Es gibt eine Reihe Ansätze, das Visitor-Muster zu verbessern oder zu erweitern [9] [13] [14] [8], die jedoch nicht unsere abstrakte Sichtweise einnehmen, eine rekursive Datenstruktur als Einheit zu betrachten. Teilweise opfern diese Ansätze Typsicherheit für Flexibilität.

Aus der funktionalen Programmierung ist eine Vielzahl von Arbeiten bekannt, die Programmtransformation nutzen um die Effizienz von Programmen zu steigern. Wir gehen davon aus, dass die Verwendung von Generic-Factories Ansatzpunkte für weitere Optimierungen liefern wird, die in normalen objektorientierten Programmen nicht bestehen. Shortcut Deforestation zeigt, dass generische Programmierung auch Laufzeitvorteile bringen kann.

Ziel laufender Forschung ist es, den Zusammenhang zwischen algebraischer Welt und objektorientierten Programmen zu formalisieren und Vorbedingungen zu identifizieren, unter denen algebraische Transformationsgesetze auf objektorientierte Programme übertragen werden können.

Literatur

1. Backhouse, Roland, Patrik Jansson, Johan Jeuring und Lambert Meertens: *Generic Programming –An Introduction–*. In: Swierstra, S. Doaitse, Pedro R. Henriques und José N. Oliveira (Herausgeber): *Revised Lectures 3rd Int. School on Advanced Functional Programming, AFP'98, Braga, Portugal, 12–19 Sept. 1998*, Band 1608 der Reihe *Lecture Notes in Computer Science*, Seiten 28–115. Springer-Verlag, Berlin, 1999.
2. Bird, Richard S.: *An Introduction to the Theory of Lists*. In: Broy, M. (Herausgeber): *Logic of Programming and Calculi of Discrete Design*, Band 36 der Reihe *NATO ASI Series F*, Seiten 3–42. Springer-Verlag, Berlin, 1987.
3. Bracha, Gilad, Martin Odersky, David Stoutamire und Philip Wadler: *Making the future safe for the past: Adding Genericity to the Java Programming Language*. In: *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Vancouver, October 1998. ACM.
4. Duris, Etienne, Didier Parigot, Gilles Roussel und Martin Jourdan: *Attribute Grammars and Folds : Generic Control Operators*. Technical Report RR-2957, Inria, Institut National de Recherche en Informatique et en Automatique.
5. Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
6. Goguen, J. A., J. W. Thatcher und E. G. Wagner: *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*. In: Yeh, Raymond T. (Herausgeber): *Current Trends in Programming Methodology*, Band 4, Seiten 80–149. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.
7. Jacobs, Bart und Jan Rutten: *A Tutorial on (Co)Algebras and (Co)Induction*. Bulletin of the EATCS, 62:222–259, 1996.
8. Krishnamurthi, Shriram, Matthias Felleisen und Danial P. Friedman: *Synthesizing Object-Oriented and Functional Design to Promote Re-use*. In: *Euro-*

pean Conference on Object-Oriented Programming (ECOOP), Seite 91ff, Brussels, July 1998.

9. Lieberherr, Karl J. und Doug Orleans: *Preventive Program Maintenance in Demeter/Java.* In: *Proceedings of the 19th International Conference on Software Engineering*, Seiten 604–605. ACM Press, Mai 1997.
10. Meertens, Lambert: *Calculate Polytypically!* In: Kuchen, H. und S. D. Swiestra (Herausgeber): *Proceedings 8th Int. Symp. on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany, 24–27 Sept 1996*, Band 1140 der Reihe *Lecture Notes in Computer Science*, Seiten 1–16. Springer-Verlag, Berlin, 1996.
11. Meinke, K. und J. V. Tucker: *Handbook of Logic in Computer Science*, Band 1, Kapitel Universal Algebra, Seite 189ff. Oxford Science Publications, 1992.
12. Odersky, Martin und Philip Wadler: *Pizza into Java: Translating theory into practice.* In: *24'th ACM Symposium on Principles of Programming Languages*, Paris, January 1997.
13. Ovlinger, Johan und Mitchell Wand: *A Language for Specifying Recursive Traversals of Object Structures.* In: *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, Seiten 70–81, November 1999.
14. Palsberg, J. und C.B. Jay: *The Essence of the Visitor Pattern.* Technischer Bericht 05, 1997. COMPSAC'98, to appear.
15. Takano, Akihiko und Erik Meijer: *Shortcut Deforestation in Calculational Form.* In: *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, Seiten 306–313. ACM Press, New York, 1995.
16. Wadler, Philip: *Theorems for free!* In: *4'th International Conference on Functional Programming Languages and Computer Architecture*, London, UK, September 1989.
17. Wadler, Philip: *Monads for functional programming.* In: Jeuring, J. und E. Meijer (Herausgeber): *Advanced Functional Programming*, Band 925 der Reihe *LNCS*. Springer Verlag, 1995.
18. Wadler, Philip: *The marriage of effects and monads.* In: *International Conference on Functional Programming*, Seiten 63–74, Baltimore, September 1998. ACM.

Distributed Programming in Haskell: From Ports to Streams

Volker Stolz and Frank Huch*

Lehrstuhl für Informatik II
RWTH Aachen
D-52056 Aachen, Germany

Abstract. We present an extension of the lazy functional programming language Haskell for distributed programming. For the communication between processes we modify the port concept to a stream based approach. Ports are accessible over the Internet and behave like channels in Concurrent Haskell except that only the process which creates a port can read from it. This restriction is not enforceable at compile time and thus may yield undesired effects at runtime. By eliminating the need for explicit `readPort` statements and instead providing a stream of the items received on a port, we can guarantee safety in regard to this matter.
The implementation consists of a library written entirely in Haskell which provides functions for creating new processes, communication between concurrent and distributed processes in an open system using ports or streams, and error handling with exceptions.

1 Distributed Programming

The development of software systems has changed in the last years. Many systems are distributed, because of the following reasons:

- **Parallelization:** Resources (e.g. speed or space) needed for an application are not sufficing on one computer.
- **Inherent distributed character:** The application itself is distributed. Examples are (mobile) telephones and a cash dispenser together with the bank server.
- **Reliability and fault tolerance:** To increase the reliability of a system it is possible to arrange for several computers to cooperate such that the failure of one or more computers does not effect the system behavior as a whole.
- **Access to special resources:** In a heterogeneous network, special resources, e.g. a scanner or printer can only be accessed from one computer.

With the boom of networks and the Internet, the number of distributed applications increases. In particular more and more applications have an inherent distributed character. To provide convenient means for programming, modern languages must support distributed programming. It is not sufficient to provide a library for communication via sockets.

The language has to supply high level concepts for distribution and communication between processes. We want to extend the functional programming language Haskell [8] with features for elegant distributed programming.

* {stolz,huch}@i2.informatik.rwth-aachen.de

W. Goerigk, E. Pulvermüller, A. Speck (Hrsg.): Sprachen, Architekturen und neue objektorientierte Softwaretechniken. Bericht Nr. 2018, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

2 Distributed Communication

Modern languages need to provide suitable ways for communication in a distributed system. Different methods like *remote procedure call* or *message passing* are employed. Standards like CORBA [10] define methods for interchanging information between different platforms.

Functional languages have been extended for concurrent and distributed programming, too. The most successful one is Erlang [1]. Unlike Haskell, Erlang is an eager functional programming language which is extended with special features for concurrent and distributed programming. Processes can be created dynamically with `spawn` on a local or even a remote computer. Every process has a process identifier (*pid*) which is used as a reference for the communication between processes. Other processes can send messages to this pid.

Arriving messages are stored in a per-process mailbox in order of their arrival (FIFO). The receiving process can conveniently access this mailbox with pattern matching, so it does not need to extract messages in their chronological order. Only relevant messages are fetched with pattern matching while the others reside in the mailbox and can be processed later.

Another important feature of Erlang is that communication between concurrent processes on the same computer is not distinguished from communication to remote processes in a network. Therefore a system developed in a concurrent setting can later be distributed easily. Scalability of the system is supported by the language.

For fault tolerant programming Erlang provides a linking mechanism. Processes of the system can automatically be informed if others die or become unreachable, for example because a computer crashes. Hence these processes can react on the failure and reorganize the system to a consistent state.

Porting this communication scheme from Erlang to Haskell proved infeasible ([6] & [13]) because of limitations in Haskell's type system. Subtyping is necessary because different processes can understand different messages, but have common subsets. Unluckily, this can't be expressed in Haskell where constructors must be unique.

Let's assess the currently available features on which a more comfortable programming environment can be based: Concurrent Haskell [9] is state of the art for concurrent programming in Haskell. It provides functions to start and terminate threads dynamically inside an application and to synchronize them with mutable variables (*MVar*). This requires leaving the purely functional context and entering the IO monad which is used for conventional input/output.

On top of these MVars semaphores and channels provide more comfortable means of message passing. But there is yet no concept for distributed programming. One possible approach is the extension of Concurrent Haskell with communication via channels and MVars for distributed programming. But this leads to implementation problems. In Concurrent Haskell many processes may synchronize on a mutable variable or a channel. Distributing some of these processes in a network leads to synchronization problems, because it is not clear where a MVar is located. Consider the situation in Figure 1. Two processes can write a mutable variable and two other processes want to read it. In a distributed setting these processes could be located on four different computers in a network. But

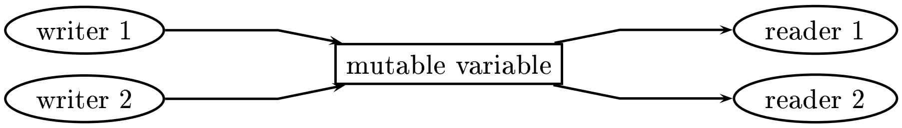


Figure 1: Distribution of Concurrent Haskell Processes

where can the mutable variable be located? It has to be located on one computer in the network, because it needs a state for the storage of a value, if no reader suspends on it and no reader wants to read the value. The possibilities for the location of the MVar are one of the four computers of the example or an independent computer. But all these locations have disadvantages for providing fault tolerance. It is necessary that parts of the system may terminate or even crash without effecting the rest of the system. If the computer the mutable variable is located on crashes, then the whole system cannot work anymore, although there is still a writer and a reader which could communicate with each other from the logical structure of the system. The readers and the other writers hang up and it is difficult to repair the system to a consistent state where the other components can communicate with each other again.

Another problem is garbage collection in a distributed system. We do not know when a mutable variable or channel is garbage. Reference counting is not possible because it would involve a large network overhead. So it will be almost impossible to check if a MVar is still known somewhere in the network and an algorithm would be very expensive and produce much communication in the network. Especially in case that some parts of the system have already crashed, some kind of polling **all** nodes which have a registered reference on the MVar would be necessary.

Systems like Glasgow parallel Haskell [14] or Glasgow distributed Haskell [11] implement distributed garbage collection, but are limited to closed systems. It is also not possible to implement fault tolerance in these systems, which is in our view a major requirement for a programming language for distributed system.

Our solution to the problems arising from distributed garbage collection is to restrict communication to only one reader for each channel. We can thus place the "physical" object onto the machine/processor which is reading it.

3 Distributed Haskell

In the discussion of distributing objects similar to Concurrent Haskell in a network, we have seen which problems appear with multiple readers and writers of a MVar. Our solution to this problem is a restriction to only one reader. With this restriction we can locate the MVar at the same place where the reader is located. If the reader terminates or crashes, the MVar terminates, too. No other readers can suspend on it. Therefore no processes are hanging, although there may still be references to the crashed resource on remote hosts. There can still exist writers that want to write to the MVar. Hence they are also in an inconsistent state. But they can recognize this the next time they send a message to the MVar. A failed write operation can throw an exception, as can the runtime system when we provide means of checking the lifeness of MVars although they are currently unused. The exception can be caught and the writer can initialize a reorganization of the crashed components, for example on another computer, or just fail gracefully.

Although this communication concept is different from Erlang, there are still similarities: Erlang also has a single-reader restriction as each mailbox is associated with a single process which is the only one who can read from it.

In the following, we will allow multiple mailboxes called *ports* (similar to the channels of Concurrent Haskell as programs usually will work an sequences of messages instead of just single values passed on using a MVar), associated to exactly **one** process which may read the data. Of course each port may have a different type. The reader is naturally the process which creates the port, although ownership can be transferred.

However, nobody precludes the programmer from passing the port via `forkIO` to another thread or reading from it on a remote machine. As a remedy we must insert checks at runtime to see if a thread is permitted to read from the port. If we can eliminate the need for explicit reading from the port we could help the programmer in avoiding mistakes. Before we show our approach to this matter, let's give a quick overview of the current API provided for Port-based Distributed Haskell:

3.1 The Distributed Haskell Library

Ports are represented as an abstract polymorphic data type

```
data Port a -- abstract
```

where the type variable `a` represents the type of the values that can be sent to the port. A new port can be created with the function

```
newPort :: IO (Port a)
```

Like in Concurrent Haskell the operations for creating and sending have side-effects because they change the environment the program is running in. Hence they belong to the IO monad which assures sequential execution of IO actions in contrast to Eden [2] and Goffin [3].

A value can be written to and read from a port with the functions

```
writePort :: Port a -> a -> IO ()
readPort  :: Port a -> IO a
```

A port can be used in the same way as a channel in Concurrent Haskell, except that only the process which creates the port can read from it.

For writing to a port it doesn't matter where a port is located. We provide all necessary "contact"-information in the data type and impose no limitations on what thread may write (see Figure 2).

For sending messages over the network we have to encode them in a binary representation as we have no access to the underlying internal closure. In this implementation as a library for the Glasgow Haskell Compiler [4] we send messages as strings. We use the function `show` from the class `Show`. Therefore the type of the messages which can be sent with `writePort` must be an instance of the class `Show`. On the other hand a message must be reconverted from the

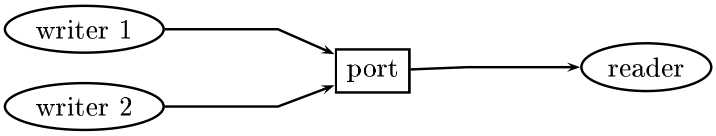


Figure 2: Modelling ports as channels in a distributed setting

string representation into the corresponding data type on reading it from a port. Therefore the type of the messages which can be received with `readPort` must be an instance of the class `Read`, too. This means the messages of a port need both instances. We subsume these properties in a new class named `Serialize`. Note that we will omit this class constraint in the type signatures for brevity.

But this imposes the restriction that no functions, infinite data structures, or mutable structures like MVars can be sent to remote threads.

For the connection of independently started components we provide a global registration mechanism. With

```
registerPort   :: Port a -> Name -> IO ()
unregisterPort :: Port a -> IO ()
```

ports can globally be registered and unregistered on one computer. Other processes can then lookup a registered port with

```
lookupPort :: Host -> Name -> IO (Port a)
```

from anywhere else in the network. In the actual implementation `Port.Host` and `Port.Name` are just type synonyms for `String`.

For suspending on two ports we provide a merge function which even allows a programmer to merge two ports of different types. We use Haskells `Either` type:

```
mergePort :: Port a -> Port b -> IO (Port (Either a b))
```

With this function a process can suspend on messages of different types. This is needed, because a process can suspend on messages from different other processes. The type of these messages should not be the same, because these processes can perform completely different tasks in the system. An example for such a system is a chat where a chat clients may receive messages from a chat server and the keyboard (respectively GUI) [7]. From the software engineering point of view a restriction to merging only ports of the same type like in Eden [2] or Goffin [3] is awful.

It should also be possible that the merged ports can afterwards also be used in their non-merged version. As the client is the only process that can read from these ports there can be no conflict that two readers want to read from the ports and their merged version at the same time.

For the creation, termination and error handling of processes we use the same functions as Concurrent Haskell and the module `Exception`:

```
forkIO        :: IO () -> IO ThreadID
myThreadId    :: IO ThreadId
killThread    :: ThreadID -> IO ()
raiseInThread :: ThreadId -> Exception -> IO ()
try           :: IO a -> IO (Either Exception a)
```

Finally we provide a linking mechanism. With the function

```
linkAndKill :: Port a -> IO Link
```

a link between the executing process and a port is established. A polling mechanism contacts the host periodically where the linked port is located and checks for its existence. If the port does no longer exist the process is terminated by an exception. The runtime system will trigger matching links, too, if it notices a transmission error. This can be caught and the process can initiate a reorganization of the whole system. A more convenient function for linking ports is the function

```
link :: Port a -> IO () -> IO Link
```

which takes an additional `IO` action as parameter. This action is performed if the linked port does not exist any more (`linkAndKill` is just an application of this function). We can use `link` for example to send a message if a port dies. Once an established link is not needed any more it is possible to remove links with the function `unlink :: Link -> IO ()`. We have also added a fault tolerant version of `writePort`

```
writePortFail :: Port a -> a -> IO () -> IO ()
```

Similar to `link` the action in the third parameter is performed in the case of an erroneous sending.

3.2 Limitations of the current port concept

In a multithreaded environment it is often necessary to allocate resources and pass them on to several threads (we refer to the notion of *threads* opposed to operating system-level *processes*). A common usage would be a simple application consisting of two threads, one writer and one reader:

```
do {p <- newPort; forkIO (reader p); writer p}
```

Although there is no apparent error in this sequence, it surely violates the single-reader criterion as specified above! We require that the only thread allowed to read is the thread which created the port. Of course this could easily be alleviated by forking the writer instead of the reader. However, we feel that the compiler should be able to reject as many faulty programs as possible. Note that distinguishing between ports using different types such as `ReadPort` and `WritePort` doesn't solve this problem, either.

In providing just a stream of messages received on a port we can avoid the necessity for an explicit `readPort` statement. If this stream is passed on to another thread like in the case above, several threads can **share** copies of this stream. Multiple threads reading one port are not competing to grab a message, either. For sending the reference to the port can still be used. We only hide `readPort` from the view of the programmer and provide a new function to create a stream with its corresponding port reference:

```
newStream :: -> IO (Port a, [a])
```

The resulting port can be used for registering or linking as before.

Another new function the API must offer is merging two streams as we can't use `mergePort` any longer. Merging has to interleave access to two streams and return a list of either an item from the first or the other stream, depending on which becomes available first.

```
mergeStreams :: [a] -> [b] -> [Either a b]
```

3.3 A Chat Example

As an example we now give the implementation of a chat server using the new stream based approach. Clients can connect to this server and messages arriving from one client will be relayed to all others. Its communication interface is given by the data type

```
data ServerMsg = Connect String (Port ClientChatMsg) | Send String String
            | Close String (Port ClientChatMsg) deriving (Read, Show, Eq)
```

The server creates a port for external communication on startup and registers this port as `ChatServer`.

```
main = do (serverPort,stream) <- newStream
          registerPort serverPort "ChatServer"
          foldM chatServer [] stream
```

After the initialization it proceeds in a loop. The loop is elegantly implemented with a `foldM` on the stream, which is an infinite list. A list of the ports of all connected clients is kept as the state of the process. This list changes in dependence of connecting or leaving clients. A new chat message is broadcasted to all the other connected clients:

```
chatServer :: [Port ClientMsg] -> ClientMsg -> IO ()
chatServer clientPorts msg = case msg of
     (Connect name clientPort) -> do
         mapM_ (\ p -> writePort p (Login name)) clientPorts
         return (clientPort:clientPorts)
     (Close name clientPort) -> do
         let newClientPorts = filter (/= clientPort) clientPorts
         mapM_ (\ p -> writePort p (Logout name)) newClientPorts
         return newClientPorts
     (Send name str) -> do
         mapM_ (\ p -> writePort p (Chat name str)) clientPorts
         return clientPorts
```

A client process uses the interface

```
data ClientChatMsg = Chat String String | Login String |
                     Logout String deriving (Read, Show, Eq)
```

to receive messages from the chat server. The `Chat` message is used for new chat messages. The first string is the nickname of the user taking part in the chat and the second string is her chat message. The `Logout` message is sent if a client leaves the chat. We don't need an algebraic data type for the interface to the keyboard process, we just use strings.

First a client process initiates the connection to a chat server on `host` by obtaining a reference to the remote stream using `lookupPort`. Then its own streams are created and a process for the input from the keyboard is forked.

```
main = do
  putStrLn "Host of chat server? "
  host <- getLine
  putStrLn "Nickname? "
  name <- getLine
  serverPort <- lookupPort host "ChatServer"
  (chatPort,chatStream) <- newStream
  writePort serverPort (Connect name chatPort)
  (keyboardPort, keyboardStream) <- newStream
  forkIO (readKeyBoard keyboardPort)
  let stream = mergeStreams chatStream keyboardStream
  mapM_ (client name serverPort)
        (takeWhile (/= (Right "")) stream)
  writePort serverPort (Close name chatPort)
```

The two streams are merged into one and the process proceeds in a loop analyzing the messages. Here we use `mapM_` instead of `foldM_` because there is no state involved in the recursion (apart from IO actions). To terminate the recursion we use `takeWhile` on the infinite stream. The empty input from the keyboard represented by the message `(Right "")` in the merged stream terminates the client. The client sends a `Close` message to the server and terminates. All other messages are handled in the function `client`. Messages from the server are displayed (not shown) and messages from the keyboard are forwarded to the server.

```
client :: String -> Port ServerMsg -> Either ServerMsg String -> IO ()
client name serverPort msg = case msg of
     (Left serverMsg) -> putStrLn (display serverMsg)
     (Right str)      -> writePort serverPort (Send name str)
```

The process for reading from the keyboard is not presented here. It just reads strings from the keyboard, sends them to the client process and terminates itself if the user inputs the empty string.

3.4 Fault tolerance

This chat application does not behave fault tolerant. If a client dies and the next chat message is broadcasted to all chat clients an exception is thrown and the chat server crashes because the port of the dead client does not exist anymore. With the linking mechanism and the use of `writePortFail` instead of `writePort` we can easily guarantee fault-tolerance for our server. In the case of a failure we close the port to which writing failed. Therefore we just have to add the following link into the `Connect` case in the server process:

```
(Connect name clientPort) -> do
   link clientPort (writePort serverPort (Close clientPort))
```

and modify the `writePort` instruction in the broadcast of chat messages:

```
writePortFail p ... (writePort serverPort (Close p)) ...
```

The chat server just sends a `Close` message to itself if a port does not exist any more (note that we have to change the loop to pass `serverPort`, too!).

4 Implementation

As Port-based Distributed Haskell is the foundation on which we built streams, we will give only a short overview of the internals of communication using ports, then focusing on the implementation of streams. A more detailed description of the layout of the internal data structures used for ports can be found in [7].

Communication via ports uses two different kinds of techniques for passing data: In case we just want to send a message to a port located in the same program, we can use a simple Concurrent Haskell channel to deliver the message. This channel is typed over the messages the port can receive.

On the other hand, we have to contemplate communication over the network. We use standard TCP/IP system-level functions to connect a socket to a remote host, marshal the message using the class `Serialize` and deliver it by writing to the file descriptor obtained from the socket. All necessary data like host name and TCP/IP port number are encoded in the port's data structure. These are used by remote hosts when writing to a port reference, e.g obtained by receiving a message over the network.

A separate program (called the "external post office") running on remote hosts which wish to offer services based on registered port names translates the symbolic names used in `registerPort` and `lookupPort` statements to their internal representation. It is necessary that this program binds to a well-known TCP/IP-port so it can be contacted without any prior knowledge. The service offered is similar to the mechanism employed in *remote procedure calls* [12].

Internally, data arriving on the socket of a port is transferred to a helper thread using a channel of type `String`. There it is converted from a string to it's equivalent Haskell expression using once again the functions provided by the class `Serialize`. Messages which can't be converted are rejected as bogus, valid ones are finally passed into the typed channel we already used for internal communication, thus forming the final per-thread message queue.

The whole structure of a port and the internal and external communication is summarized in Figure 3.

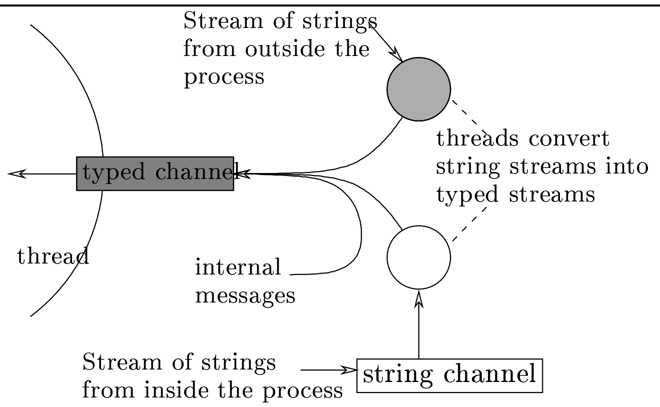


Figure 3: Structure of a port and internal/external communication

A polling mechanism is employed for the linking of ports. All linked ports are stored in a database. A process in the background polls all registered ports in a fixed schedule. If one of these ports does not exist anymore, an exception is thrown (`linkAndKill`) or the specified IO action is performed (`link`, `writePortFail`). Of course the latter could be programmed by hand from the programmer using `catch`. The implementation of links needs the Distributed Haskell runtime system. They cannot be handled outside the Distributed Haskell library.

4.1 Streams

Although streams provide a powerful abstraction they are yet easy to implement. The only major change related to the `Port` module is that we need a version of `readPort` without the safety belt which checks the `ThreadID` of the caller. The helper function `getPortContents` will retrieve all items in a port in a list. This function is modeled in the same fashion as `getChanContents` from the module Concurrent. This includes suspending until a new message arrives in case the port is empty.

```
getPortContents :: Serialize a => Port a -> IO [a]
newStream       :: Serialize a => IO (Port a, [a])
```

Merging two streams is much more difficult. We have to use two different threads, each one waiting for messages on its stream and forwarding them encapsulated in an `Either` type to a common channel.

```
mergeStreams :: [a] -> [b] -> [Either a b]
```

5 Related Work

– **Goffin** [3] extends Haskell with concurrent constraint programming and a special port concept for internal and external communication. The ports are not integrated in the IO monad.

For reacting on multiple ports Goffin proposes a fair `merge :: Port a -> Port a -> Port a` for ports. But both merged ports must have the same type. This restriction is too strong as we have seen in the chat example where the `Either` type was essential.

– **Eden** [2] is an extension of Haskell for concurrent and parallel programming. A process concept is added in which every process has a fixed number of input and output channels for communication with other processes. Communication is not integrated in the IO monad and with a fair `merge`, which is part of Eden, processes can behave nondeterministically. Like in Goffin this `merge` is restricted to channels of the same type.

Furthermore in Eden a process can only read from or write to a fixed number of channels. The connections between the processes cannot be changed dynamically.

Eden is developed for parallel programming where programs have a more hierarchical structure than in distributed programming and it is difficult to implement complex protocols in Eden. It is also not possible to connect two independently started processes in Eden.

– **Curry** [5] is a functional logic programming language which extends Haskell with needed narrowing, residuation and encapsulated search. For distributed programming Curry adds named ports which guarantee that all readers of one port are executed in the same Curry program. This is thought to eliminate the implementation problems with multiple readers. On the other hand a programmer can also send logical variables through ports which is proposed as an easy answering mechanism. But with these logical variables channels which have multiple readers can be programmed. This results in the same problems with multiple readers which should be avoided by the introduction of ports. Logical variables can only be used as comfortable answer variables. This is no restriction from the formal semantics, but from the implementation.

Communication in Curry is like in Goffin a constraint which has to be solved. A programmer must learn a new programming paradigm. Furthermore only the external communication must be integrated in the `IO` monad. Internally concurrent processes communicate via lazy streams. This can result in problems with laziness and strictness annotations have to be added sometimes. Another problem is that a concurrent application cannot easily be distributed to a network because the processes have to be transfered into the IO monad. This can yield problems with the scalability of a system.

– **Glasgow distributed Haskell** [11] is an approach for the integration of Glasgow parallel Haskell [14] and Concurrent Haskell. It provides a closed distributed systems and communication between processes like in Concurrent Haskell. The main idea is the distribution of a shared memory system. Communication between processes is not strict. Hence the programmer does not know when data is exchanged between the components of the network and can not estimate when and how much net traffic is produced.

Fault tolerance is restricted to error handling like in Concurrent Haskell. If one of the computers in the network crashes, then the whole system crashes, too. But the greatest disadvantage of this approach is the restriction to closed systems which makes it impossible to implement many distributed applications like telephony or peer-to-peer applications.

– Finally, we once again want to compare our approach with **Erlang** [1]. The first advantage of Distributed Haskell is that messages are statically typed. The type of the messages which can be sent through a port are the communication interface of the process which reads from the port. In Erlang such an interface does not exist. Furthermore this type system provides safety in program development. For example typos yield a compile-time error in Distributed Haskell, not a deadlock like in Erlang. Another advantage is our linking mechanism which is more powerful than linking in Erlang. We can add arbitrary `IO` actions to the links which are performed if the linked port dies. In Erlang it is only possible to receive a message if another process dies.

6 Conclusion and Future Work

Our modification of the port-based approach introduces the notion of streams. Streams are basically a list of the messages received on a typed port. Both concurrent and distributed programming is covered by this approach. The limitations imposed by the single-reader semantics of ports disappear when we abstract from the explicit `readPort` statement.

All described extensions are implemented using the Glasgow Haskell Compiler [4], based on the libraries `Concurrent` and `Socket`. The module `PStream` can be used together with any Haskell program compiled using the Glasgow Haskell Compiler. The library sources and sample applications can be found at `http://www-i2.informatik.rwth-aachen.de/hutch/DistributedHaskell/`.

A possible future focus will be on analyzing how good distributed programming and lazy evaluation match. External communication forces eager evaluation of large parts of a program. Furthermore, usually other IO actions will require a serial execution, thus reducing the advantage of lazy evaluation, too. As we can't yet send heap fragments to remote hosts, the advantages of lazy functional programming may be limited to the "computational" (in contrast to the "communication") part of programs. On the other hand the higher order functions used for working on streams provide a good abstraction from the otherwise tedious work of explicitely programming the message exchange.

References

1. J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
2. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Pena-Mari. Eden — The paradise of functional concurrent programming. *LNCS*, 1123:710ff., 1996.
3. Manuel M. T. Chakravarty, Yike Guo, and Martin Köhler. Distributed Haskell: Goffin on the Internet. In M. Sato and Y. Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, pages 80–97. World Scientific Publishers, 1998.
4. The Glasgow Haskell compiler. http://www.haskell.org/ghc/.
5. M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99).* Springer LNCS (to appear), 1999.
6. Frank Huch. Erlang-style Distributed Haskell. In *Draft Proceedings of the 11th International Workshop on Implementation of Functional Languages*, September 7th – 10th 1999.
7. Frank Huch and Ulrich Norbisrath. Distributed programming in Haskell with ports. *LNCS*, 2011:107–121, 2000.
8. Simon Peyton Jones et al. Haskell 98 report. Technical report, http://www.haskell.org, 1998.
9. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 January 1996.
10. Object Management Group. Common Object Request Broker Architecture (CORBA). http://www.omg.org/gettingstarted/.
11. R. Pointon, P. Trinder, and H-W. Loidl. The design and implementation of glasgow distributed Haskell. *LNCS*, 2011:53–70, 2000.
12. Richard W. Stevens. *UNIX Network Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.
13. Volker Stolz. Robuste verteilte Programmierung in Haskell, 2001. RWTH Aachen, Germany.
14. Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.