Neuantrag bei der DFG auf
Gewährung einer Sachbeihilfe für das Projekt

Eine generische funktionale Programmiersprache: Theorie, Sprachentwurf, Implementierung und Anwendung

Antragsteller
Hochschuldozent Dr. Ralf Hinze
Universität Bonn

Juli 2004

1 Allgemeine Angaben

Neuantrag auf Gewährung einer Sachbeihilfe

1.1 Antragsteller

Name:	Dr. Ralf Hinze		
Dienststellung:	Hochschuldozent (C2)		
Geburtsdatum:	2. Juli 1965		
Nationalität:	deutsch		
Institution:	Institut für Informatik III Universität Bonn		
Dienstadresse:	Institut für Informatik III Universität Bonn Römerstraße 164 53117 Bonn	Privatadresse:	Im Obstgarten 6 53639 Königswinter (02244) 871111
Telefon:	(0228) 73 4535		
Telefax:	(0228) 73 4382		
Email:	`ralf@informatik.uni-bonn.de`		

1.2 Thema

Eine generische funktionale Programmiersprache: Theorie, Sprachentwurf, Implementierung und Anwendung

1.3 Kennwort

„GFP“

1.4 Fachgebiet und Arbeitsrichtung

- Praktische Informatik
  - Programmiersprachen
    - Funktionale Programmiersprachen

* Übersetzerbau
* Typsysteme
* Programmverifikation

1.5 Voraussichtliche Gesamtdauer

- Voraussichtliche Gesamtdauer des Vorhabens: 3 Jahre.
- Erforderliche Förderung durch die DFG: 3 Jahre.

1.6 Antragszeitraum

36 Monate

1.7 Gewünschter Beginn der Förderung

1. Oktober 2004

1.8 Zusammenfassung

Funktionale Programmiersprachen sind Vorreiter für statische Typsysteme. Statische Typen dienen der Vermeidung von Laufzeitfehlern, indem zur Entwicklungszeit sichergestellt wird, dass Programme ihre Daten in konsistenter Weise verwenden.

Je mehr Eigenschaften der verwendeten Daten im Typsystem erfasst werden, desto zuverlässiger arbeitet ein Programm. Eine Vielzahl von verwendeten Datentypen steht jedoch der Wiederverwendbarkeit von Programmcode entgegen: Die meisten Operationen lassen sich weder auf alle Datentypen in gleicher Weise übertragen, noch sind sie völlig unabhängig von der Natur des zugrundeliegenden Typs. Alle Datentypen haben jedoch eine gemeinsame Struktur (Record- und Union-Typen gepaart mit Rekursion), und wenn man erlaubt, über diese Struktur zu abstrahieren, kann man generische Programme schreiben, die sich für eine Vielzahl von Datentypen wiederverwenden lassen und die sich bei Änderungen von Datentypen automatisch an die neuen Gegebenheiten anpassen.

Da die Entwicklung auf dem Gebiet der Typsysteme große Fortschritte macht und zu erwarten ist, dass die formale Verifikation von Programmen weiterhin an Bedeutung gewinnt, wird auch die generische Programmierung zunehmend unentbehrlich. Wir wollen bisherige, sprachspezifische ad-hoc-Lösungen auf eine allgemeine Basis stellen, ohne dabei die Implementation aus den Augen zu verlieren: Wie soll eine ideale generische Programmiersprache aussehen? Wie kann sie effizient implementiert werden? Welche bekannten Anwendungen lassen sich effizienter oder eleganter lösen und welche neuen Anwendungen erschließen sich?

2 Stand der Forschung, eigene Vorarbeiten

2.1 Stand der Forschung

Begrifflichkeit Das Konzept der generischen funktionalen Programmierung firmiert unter einer Reihe von Namen: *Strukturpolymorphismus* [Ruehr 1992, Ruehr 1998], *typparametrische Programmierung* [Sheard 1993], *Gestaltpolymorphismus* (engl. shape polymorphism) [Jay und Cocket 1994], *intensionaler Typpolymorphismus* (engl. intensional type analysis) [Harper und Morrisett 1995a], *extensionaler Typpolymorphismus* [Dubois et al. 1995], *polytypische Programmierung* [Jeuring und Jansson 1996] und *daten-generische Programmierung* [Gibbons 2003]. Da generische Funktionen von der Struktur von Typen abstrahieren, charakterisiert der Begriff „Strukturpolymorphismus“ das Konzept unseres Erachtens am besten. Gleichwohl hat sich der allgemeinere Terminus „generische Programmierung“ in den letzten Jahren durchgesetzt. Der Begriff der generischen Programmierung wird auch in der objekt-orientierten Programmierung verwendet [Garcia et al. 2003], meint dort aber etwas gänzlich anderes: parametrisierten Typpolymorphismus à la Hindley [Hindley 1969] und Milner [Milner 1978].

Geschichte Die ersten Arbeiten zur generischen Programmierung basieren auf der Interpretation von Datentypen als initiale Algebren mit zugehörigen *map*-Funktionen und Katamorphismen [Meijer et al. 1991, Bird und de Moor 1997]. Die Programmiersprache *Charity* [Cockett und Fukushima 1992] stellt diese Funktionen automatisch für jeden benutzerdefinierten Datentyp zur Verfügung. Die Sprache ist *stark normalisierend*, daher verfügt sie auch nicht über allgemeine Rekursion. *Functorial ML* [Jay et al. 1998] bietet eine ähnliche Funktionalität, basiert aber auf der Theorie des Gestaltpolymorphismus (Daten werden separiert in Form und Inhalt). *PolyP* [Jansson und Jeuring 1997] erweitert die Programmiersprache Haskell um ein spezielles Sprachkonstrukt zur Definition generischer Funktionen. Alle genannten Ansätze sind auf reguläre Datentypen erster Stufe eingeschränkt, das heißt, sie decken nur einen kleinen Teil der (zum Beispiel in) Haskell definierbaren Datentypen ab. So genannte „nested data types“ [Bird und Meertens 1998] und Datentypen höherer Ordnung [Hinze 2001a] (siehe auch Abschnitt 2.1.3), die allesamt in Haskell zulässig sind, bleiben bei diesen Ansätzen außen vor.

2.1.1 Haskells Typklassen

Herausragendstes Merkmal der Sprache Haskell [Peyton Jones 2003] im Vergleich zu anderen statisch getypten Programmiersprachen ist das Konzept der *Typklassen* [Hall et al. 1996]. Typklassen ermöglichen es, den gleichen Namen für verschiedene, „typähnliche“ Funktionen

zu verwenden (*overloading*). Der Gleichheitstest ist in Haskell zum Beispiel als Klassenmethode vordefiniert.

$$\begin{array}{l}\textbf{class } \mathit{Eq}\ \alpha \textbf{ where} \\ \quad (==) \ :: \ \alpha \to \alpha \to \mathit{Bool}\end{array}$$

Soll der Gleichheitstest für einen bestimmten Typ verwendet werden, so muss dieser Typ zu einer Instanz der Klasse *Eq* gemacht werden. Die folgende Instanz definiert die Gleichheit von Listen:

$$\begin{array}{l}\textbf{instance } (\mathit{Eq}\ \alpha) \Rightarrow \mathit{Eq}\ [\alpha] \textbf{ where} \\ \quad \begin{array}{lcl}[\,] == [\,] & = & \mathit{True} \\ [\,] == (a_2 : as_2) & = & \mathit{False} \\ (a_1 : as_1) == [\,] & = & \mathit{False} \\ (a_1 : as_1) == (a_2 : as_2) & = & (a_1 == a_2) \wedge (as_1 == as_2)\end{array}\end{array}$$

Typklassen sind in gewisser Hinsicht Vorläufer generischer Definitionen: eine generische Funktion arbeitet automatisch auf *allen* algebraischen Datentypen, wohingegen eine Klassenmethode explizit für jeden neuen Typ von Hand programmiert werden muss. Eine Ausnahme stellen die so genannten *ableitbaren Klassen* dar: für eine Handvoll vordefinierter Klassen, unter anderem für *Eq*, können vom Übersetzer automatisch Instanzen erzeugt werden.

$$\textbf{data } \mathit{Tree}\ \alpha \ = \ \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha) \textbf{ deriving } (\mathit{Eq})$$

Die **deriving**-Klausel instruiert den Übersetzer, eine *Eq* Instanz für den Datentyp *Tree* zu erzeugen. Eine Verallgemeinerung des **deriving**-Mechanismus auf beliebige benutzerdefinierte Klassen mittels Methoden der generischen Programmierung ist in dem Artikel „Derivable type classes“ [Hinze und Peyton Jones 2001] beschrieben—der Vorschlag ist teilweise im Glasgow Haskell Compiler umgesetzt.

2.1.2 Intensional type analysis

„Intensional type analysis“ (ITA) [Harper und Morrisett 1995b, Weirich 2001] wurde entwickelt, um die Implementierung polymorpher Funktionen zu optimieren: eine polymorphe Funktion erhält ein zusätzliches Typargument, das *zur Laufzeit* analysiert werden kann (unter Verwendung eines **typecase**-Konstrukts), um etwa effizienteren typspezifischen Code auszuführen. Sprachen, die auf ITA basieren, sind für die Verwendung in Übersetzern konzipiert und nicht als Medium für Programmierer gedacht. Gleichwohl zeigen aktuelle Arbeiten [Weirich 2002, Vytiniotis et al. 2004], dass enge Verbindungen zur generischen Programmierung bestehen.

2.1.3 Generic Haskell

Generic Haskell [Clarke et al. 2001, Clarke et al. 2002] ist eine Erweiterung von Haskell um generische Sprachkonstrukte, die auf den Arbeiten des Antragstellers basiert und im Rahmen des „Generic Haskell“ Projektes an der Universität Utrecht entwickelt wurde. Der Antragsteller hat das Projekt mit initiiert und während eines einjährigen Forschungsaufenthaltes in Utrecht wissenschaftlich begleitet.

Generic Haskell erlaubt im Unterschied zu den bisher vorgestellten Ansätzen die Definition generischer Funktionen, die auf *allen* algebraischen Datentypen arbeiten. Insbesondere sind generische Funktionen auch auf *Typkonstruktoren höherer Ordnung* anwendbar. In Haskell werden Typkonstruktoren mit Hilfe so genannter „kinds“ (deutsch: Sorten) klassifiziert: Typen wie *Int* oder *Bool* besitzen die Sorte $\star$, Typkonstruktoren der Sorte $\kappa_1 \to \kappa_2$ bilden Typkonstruktoren der Sorte κ_1 auf Typkonstruktoren der Sorte κ_2 ab. In anderen Worten, Haskells Typsystem hat im wesentlichen die Ausdruckskraft des einfach getypten λ-Kalküls. Aus diesem Grund hängt der Typ einer generischen Funktion nicht nur vom Typargument, sondern auch von dessen Sorte ab. Der generische Gleichheitstest *equal* besitzt zum Beispiel den Typ (die spitzen Klammern schließen Typ- bzw. Sortenargumente ein):

$$equal\,\langle \tau :: \kappa \rangle \quad :: \quad Equal\,\langle \kappa \rangle\ \tau$$

wobei *Equal* durch Induktion über die Struktur von Sorten definiert ist.

$$\begin{array}{lcl} \textbf{type}\ Equal\,\langle \star \rangle\ \tau & = & \tau \to \tau \to Bool \\ \textbf{type}\ Equal\,\langle \kappa_1 \to \kappa_2 \rangle\ \tau & = & \forall \alpha\,.\,Equal\,\langle \kappa_1 \rangle\ \alpha \to Equal\,\langle \kappa_2 \rangle\ (\tau\ \alpha) \end{array}$$

Der Typkonstruktor *Tree* (siehe 2.1.1) hat die Sorte $\star \to \star$; somit ergibt sich für die Spezialisierung von *equal* auf *Tree* folgender Typ:

$$equal\,\langle Tree :: \star \to \star \rangle \quad :: \quad \forall \alpha\,.\,(\alpha \to \alpha \to Bool) \to (Tree\ \alpha \to Tree\ \alpha \to Bool)$$

Die Signatur macht explizit: um zwei Bäume auf Gleichheit zu testen, wird ein Gleichheitstest auf den Knotenmarkierungen benötigt, der als erster Parameter übergeben werden muss.

Die Definition von *equal* selbst ist denkbar einfach. Es genügt, drei Instanzen zu spezifizieren: eine Instanz für den einelementigen Typ 1, eine für binäre Summen und eine für binäre Produkte.

$$\begin{array}{lcl} equal\,\langle 1 \rangle\ ()\ () & = & True \\ equal\,\langle + \rangle\ eqa\ eqb\ (Inl\ a_1)\ (Inl\ a_2) & = & eqa\ a_1\ a_2 \\ equal\,\langle + \rangle\ eqa\ eqb\ (Inl\ a_1)\ (Inr\ b_2) & = & False \\ equal\,\langle + \rangle\ eqa\ eqb\ (Inr\ b_1)\ (Inl\ a_2) & = & False \\ equal\,\langle + \rangle\ eqa\ eqb\ (Inr\ b_1)\ (Inr\ b_2) & = & eqb\ b_1\ b_2 \\ equal\,\langle \times \rangle\ eqa\ eqb\ (a_1, b_1)\ (a_2, b_2) & = & eqa\ a_1\ a_2 \wedge eqb\ b_1\ b_2 \end{array}$$

Da alle algebraischen Datentypen Summen von Produkten (gepaart mit Rekursion) entsprechen, kann ausgehend von dieser Definition der Gleichheitstest für jeden neuen Datentyp automatisch erzeugt werden.

Generic Haskell unterstützt nicht nur generische Funktionen, sondern auch *generische Typen* [Hinze et al. 2004], die an die Struktur eines Typarguments angepasst sind. Paradigmatisches Beispiel für einen generischen Typ sind *digitale Suchbäume* oder „tries“ [Hinze 2000b], in denen die Struktur des Typs der Schlüssel für eine effiziente Abspeicherung der Werte genutzt wird.

$$\begin{array}{lcl} \textbf{data}\ \mathit{Trie}\ \langle 1\rangle\ \nu & = & \mathit{Maybe}\ \nu \\ \textbf{data}\ \mathit{Trie}\ \langle +\rangle\ \tau_\alpha\ \tau_\beta\ \nu & = & (\tau_\alpha\ \nu, \tau_\beta\ \nu) \\ \textbf{data}\ \mathit{Trie}\ \langle \times\rangle\ \tau_\alpha\ \tau_\beta\ \nu & = & \tau_\alpha\ (\tau_\beta\ \nu) \end{array}$$

Der Suchbaum $\mathit{Trie}\ \langle\alpha\rangle\ \nu$ realisiert eine endliche Abbildung von α, dem Typ der Schlüssel, auf ν, dem Typ der assoziierten Werte, und ist in Abhängigkeit vom Schlüsseltyp α definiert.

Generic Haskell ist in mehreren Anwendungen erprobt worden [Hinze und Jeuring 2003a]: digitale Suchbäume, Kompression von XML-Dokumenten, grundlegende Datenstrukturen für Strukturediteren. Die praktischen Erfahrungen haben jedoch auch Schwächen des Ansatzes aufgezeigt. Verschränkt rekursive generische Funktionen sind auf Grund der Form der Definitionen nur mühsam zu realisieren; verschränkte Rekursion ist aber gerade für die Programmierung nicht-trivialer Funktionen auf generischen Typen notwendig. Typspezifisches Verhalten lässt sich nicht erreichen: oft benötigt man eine Variante einer generischen Funktion, die sich für einen speziellen Datentyp oder einen speziellen Konstruktor anders verhält.

2.1.4 Generic Clean

Generic Haskell ist als Präprozessor realisiert, der generische Programme mittels Spezialisierung in „normale“ Haskell-Programme übersetzt. Die Programmiersprache Clean bietet ebenfalls eine generische Spracherweiterung an [Alimarine und Plasmeijer 2001]; diese ist allerdings vollständig in die Sprache integriert. Generische Funktionen werden über das Klassensystem eingeführt—dem sortenindizierten Typ *Equal* in Generic Haskell entspricht in Clean eine sortenindizierte Familie von Klassen. Instanzen generischer Klassen können für neu eingeführte algebraische Datentypen automatisch erzeugt werden. Die Verwendung des Klassensystems hat den Vorteil, dass Typargumente generischer Funktionen in der Regel nicht explizit angegeben werden müssen (gleichwohl muss die *Sorte* des Typarguments spezifiziert werden, damit die korrekte Instanz aus der sortenindizierten Klassenfamilie selektiert werden kann).

Generic Clean ist bereits erfolgreich in verschiedenen Anwendungen eingesetzt worden, etwa für die Generierung von Testdaten [Koopman et al. 2003] oder für die automatische Erzeugung grafischer Benutzungsoberflächen [Achten et al. 2004]. Weiterhin gibt es erste Ansätze zur Optimierung generischer Funktionen [Alimarine und Smetsers 2004].

Da Generic Clean auf den gleichen Grundlagen wie Generic Haskell basiert, sind auch die Schwächen im wesentlichen die von Generic Haskell. Darüber hinaus werden generische Typen nicht unterstützt.

2.1.5 Stratego

Stratego [Visser 2000] ist eine Spezialsprache für Programmtransformationen. Elementare Transformationen können in Stratego mit Hilfe von Kombinatoren zu komplexen Strategien verknüpft werden. Stratego unterstützt sowohl generische Transformationen, die auf allen Datentypen arbeiten, als auch spezifische Transformationen, die auf einen bestimmten Datentyp oder Konstruktor eingeschränkt sind.

Die Idee der kombinatorbasierten generischen Programmierung wurde inzwischen auf Haskell adaptiert [Lämmel und Peyton Jones 2003, Lämmel und Peyton Jones 2004]. Eine der Stärken des Ansatzes liegt in der Verarbeitung und Manipulation komplexer Datenstrukturen (etwa abstrakter Syntaxbäume oder XML-Dokumente); „klassische“ generische Funktionen wie *map* können allerdings nicht realisiert werden, ebensowenig wie generische Typen.

2.1.6 Fazit und Wertung

Mindestens drei Ansätze zur generischen Programmierung lassen sich unterscheiden:

1. hierarchischer Ansatz: generische Funktionen werden auf eine Kernsprache aufgesetzt (Typklassen, Generic Haskell und Generic Clean),
2. interpretativer Ansatz: generische Funktionen analysieren zur Laufzeit Typargumente (Intensional type analysis),
3. kombinatorbasierter Ansatz: generische Funktionen werden aus primitiven generischen Operationen zusammengesetzt (Stratego).

Jeder Ansatz hat seine spezifischen Stärken und Schwächen: der hierarchische Ansatz behandelt die größte Klasse von Datentypen, unterliegt aber einer Reihe von Einschränkungen (keine verschränkt rekursiven Funktionen oder generische Funktionen höherer Ordnung); der interpretative Ansatz ist sehr ausdrucksstark, eignet sich aber in seiner gegenwärtigen Form nicht für die Programmierung; der kombinatorbasierte Ansatz hat seine Stärke in der Verarbeitung umfangreicher Datenstrukturen, ist aber in der Ausdruckskraft eingeschränkt. In dem beantragten Projekt soll eine ideale Kombination dieser Ansätze entwickelt und theoretisch untermauert werden.

2.2 Eigene Vorarbeiten

Der Antragsteller hat die Forschung auf dem Gebiet der generischen funktionalen Programmierung durch eine Vielzahl von Beiträgen wesentlich mitbestimmt. Die grundlegenden Arbeiten zu Theorie und Implementierung sind in der Habilitationsschrift „Generic Programs and Proofs“ [Hinze 2000c] zusammengefasst. Wesentliche Teile der Schrift wurden auf der „Summer School on Generic Programming“ vorgestellt; das Begleitmaterial ist im Springer-Verlag (Lecture Notes in Computer Science 2793) veröffentlicht worden [Hinze und Jeuring 2003b, Hinze und Jeuring 2003a]. Techniken der generischen Programmierung sind darüber hinaus in dem Lehrbuch „The Fun of Programming“, Kapitel 12 [Hinze 2003], beschrieben.

Der Antragsteller ist Mitglied der IFIP Working Groups 2.1 (Algorithmic Languages and Calculi) und 2.8 (Functional Programming), Mitglied des Haskell 98 Standardisierungskomitees und einer der Hauptarchitekten von Generic Haskell.

Andres Löh, unser Wunschkandidat für die beantragte Stelle, hat sich in seiner Doktorarbeit „Exploring Generic Haskell“ [Löh 2004] (eingereicht am 10. Juni 2004) intensiv mit Erweiterungen von Generic Haskell und alternativen Ansätzen zur generischen Programmierung beschäftigt. Er ist Autor des Generic Haskell-Compilers und verfügt über reichhaltige Erfahrungen in der Anwendung generischer Techniken.

2.2.1 Theorie und Sprachentwurf

Generische Programme abstrahieren über die Struktur von Typen. Die Ausdruckskraft und Mächtigkeit verschiedener Ansätze zur generischen Programmierung hängt wesentlich davon ab, wie diese Struktur modelliert wird. Die ersten Arbeiten basierten auf der kategoriellen Interpretation von Datentypen als initiale Algebren und waren eingeschränkt auf reguläre Datentypen erster Stufe. Die Einschränkung auf reguläre Datentypen konnte aufgehoben werden, indem Datentypen als rationale Bäume [Hinze 1999c, Hinze 1999b] bzw. allgemeiner als algebraische Bäume [Hinze 1999d, Hinze 2001b] modelliert wurden. Auf diese Weise war es möglich, Funktionen über allen Datentypen der Sorte $\star \to \star$, so genannten *Funktoren*, generisch zu definieren.

Auf drei aktuelle Ansätze, die die Klasse der behandelbaren Datentypen wesentlich erweitern, wollen wir im folgenden genauer eingehen: den *rekursiven Stil* (R-Stil), den *Modell-Stil* (M-Stil), und den *Abhängigkeits-Stil* (A-Stil).

R-Stil Nicht alle generischen Funktionen abstrahieren über Funktoren. Zum Beispiel sind alle Klassenmethoden, die in Haskell „ableitbar“ sind, mit Typen der Sorte $\star$ parametrisiert. Um auch diese Methoden generisch definieren zu können, wurde der Ansatz in [Hinze 2000e] auf beliebige Datentypen erster und zweiter Stufe verallgemeinert. Generische Funktionen werden mittels Pattern-Matching auf Typmustern definiert, wobei Form und Anzahl der Typmuster von der Sorte des Typarguments abhängen. Die *map*-Funktion ist zum Beispiel durch folgende Definition gegeben (Λ bezeichnet die Typabstraktion):

$$
\begin{array}{lcl}
map\ \langle \phi :: \star \to \star \rangle & :: & (\alpha \to \beta) \to (\phi\ \alpha \to \phi\ \beta) \\
map\ \langle \Lambda\alpha\,.\,\alpha \rangle\ f\ a & = & f\ a \\
map\ \langle \Lambda\alpha\,.\,1 \rangle\ f\ () & = & () \\
map\ \langle \Lambda\alpha\,.\,\phi\ \alpha + \psi\ \alpha \rangle\ f\ (Inl\ a) & = & Inl\ (map\ \langle \phi \rangle\ f\ a) \\
map\ \langle \Lambda\alpha\,.\,\phi\ \alpha + \psi\ \alpha \rangle\ f\ (Inr\ b) & = & Inr\ (map\ \langle \psi \rangle\ f\ b) \\
map\ \langle \Lambda\alpha\,.\,\phi\ \alpha \times \psi\ \alpha \rangle\ f\ (a, b) & = & (map\ \langle \phi \rangle\ f\ a, map\ \langle \psi \rangle\ f\ b)
\end{array}
$$

Ein konkreter Sprachentwurf für eine Erweiterung von Haskell um generische Definitionen dieser Form ist in [Hinze 1999a] beschrieben.

M-Stil Ein Nachteil des obigen Ansatzes ist, dass eine generische Funktion nur auf Typen *einer* festgelegten Sorte arbeitet. Die *map*-Funktion lässt sich aber sinnvoll für Typen beliebiger Sorten definieren. Soll eine generische Funktion mit Typen unterschiedlicher Sorten parametrisiert werden, dann muss der Typ der generischen Funktion in Abhängigkeit von der Sorte des Typparameters variieren [Hinze 2000f, Hinze 2002]: die generische Funktion erhält einen sortenindizierten Typ (siehe auch 2.1.3). Der Typ von *map* lautet zum Beispiel:

$$map\ \langle \tau :: \kappa \rangle \quad :: \quad Map\ \langle \kappa \rangle\ \tau\ \tau$$

wobei *Map* durch Induktion über die Struktur von Sorten definiert ist.

$$\begin{array}{lcl} \textbf{type}\ Map\ \langle \star \rangle\ \tau_1\ \tau_2 & = & \tau_1 \to \tau_2 \\ \textbf{type}\ Map\ \langle \kappa_1 \to \kappa_2 \rangle\ \tau_1\ \tau_2 & = & \forall \alpha_1\ \alpha_2\,.\, Map\ \langle \kappa_1 \rangle\ \alpha_1\ \alpha_2 \to Map\ \langle \kappa_2 \rangle\ (\tau_1\ \alpha_1)\ (\tau_2\ \alpha_2) \end{array}$$

Die Funktion *map* selbst muss lediglich für die drei Typkonstruktoren 1, '+' und '×' definiert werden.

$$\begin{array}{lcl} map\ \langle 1 \rangle\ () & = & () \\ map\ \langle + \rangle\ mapa\ mapb\ (Inl\ a) & = & Inl\ (mapa\ a) \\ map\ \langle + \rangle\ mapa\ mapb\ (Inr\ b) & = & Inr\ (mapb\ b) \\ map\ \langle \times \rangle\ mapa\ mapb\ (a, b) & = & (mapa\ a, mapb\ b) \end{array}$$

In diesem Ansatz werden Typen als Terme des einfach getypten λ-Kalküls modelliert (die Sorten entsprechen den Typen des Kalküls). Eine generische Funktion definiert ein so genanntes *Umgebungsmodell* (der Programmierer legt die Interpretation der Typkonstanten fest). Die Spezialisierung einer generischen Funktion auf einen Typ τ entspricht dann der Interpretation von τ in diesem Modell. Dieser Ansatz ist die theoretische Grundlage sowohl von Generic Haskell als auch von Generic Clean.

Die Idee der generischen Funktionen lässt sich auch auf die Ebene von Typen übertragen [Hinze et al. 2002, Hinze et al. 2004]: so wie eine generische Funktion in Abhängigkeit von der Struktur eines Datentyps unterschiedliches Verhalten zeigen kann, ermöglichen generische Datentypen eine Implementierung, die an die Struktur des Typarguments angepasst ist (siehe 2.1.3). Viele Konzepte von generischen Funktionen lassen sich in gleicher Weise auf generische Datentypen anwenden, aber in der Kombination liegt die Stärke: Operationen auf solchen Datentypen sind von Natur aus generische Operationen.

A-Stil Wenn man R- und M-Stil miteinander vergleicht, lässt sich zusammenfassend sagen, dass Funktionen im M-Stil wesentlich allgemeiner sind, Funktionen im R-Stil sich hingegen wesentlich einfacher definieren lassen. Im R-Stil hat man Typmuster, die Variablen enthalten, und kann rekursive Aufrufe von generischen Funktionen als solche schreiben, während der M-Stil ein genaues Verständnis der zugrundeliegenden Theorie erfordert, da das Rekursionsschema implizit bleibt, und außerdem die Definition eines sortenindizierten Datentyps für jede generische Funktion verlangt.

Im „dependency style“ oder Abhängigkeitsstil (A-Stil) [Löh et al. 2003, Löh 2004] werden die Stärken der beiden Ansätze miteinander kombiniert. Syntaktisch können Funktionen wie im R-Stil geschrieben werden. Sie werden vom Übersetzer dann in den M-Stil übersetzt, so dass dessen höhere Ausdruckskraft zur Verfügung steht und die Einschränkungen des R-Stils umgangen werden. Darüber hinaus ermöglicht der A-Stil verschränkt rekursive generische Funktionen, die in den Typen der Funktionen als Abhängigkeiten festgehalten werden.

Spracherweiterungen Praktische Erfahrungen mit dem Generic Haskell-Compiler haben gezeigt, dass einige relativ häufig auftretende Programmiermuster nur mühsam zu realisieren sind. Zum Beispiel findet man sich oft in der Situation, dass man eine Variante einer bereits existierenden generischen Funktion programmieren will, die sich auf einzelnen Datentypen anders verhält als das Original. Übliche Sprachmöglichkeiten genügen nicht, denn generische Funktionen sind in der Regel rekursiv, und es gilt, auch diese rekursiven Aufrufe an die neuen Gegebenheiten anzupassen. In [Clarke und Löh 2003] ist eine Spracherweiterung beschrieben, mit der man generische Funktionen um neue Fälle erweitern und alte Fälle überschreiben kann, und zwar so, dass rekursive Aufrufe ebenfalls die neue Funktion verwenden. Zusammen mit zwei weiteren kleinen Erweiterungen erschließt sich damit ein neues Anwendungsgebiet für Generic Haskell, nämlich die Manipulation von großen Datenstrukturen. Interessanterweise haben diese Spracherweiterungen Ähnlichkeiten zu Konzepten aus der objektorientierten Programmierung. Diese Beziehung ist bislang aber noch unerforscht.

Alternative Ansätze Generische Funktionen können wie erwähnt als Verallgemeinerung von Haskells Klassen aufgefasst werden: jede Klasseninstanz muss von Hand programmiert werden, wohingegen eine generische Funktionen automatisch instantiiert werden kann. Eine naheliegende Idee ist, Klassenmethoden generisch zu definieren, so dass auch für diese automatisch Code erzeugt werden kann [Hinze und Peyton Jones 2001]. Ein Vorteil dieses Ansatzes ist die harmonische Integration von generischen Funktionen, die „structural typing“ verwenden (zwei Typen sind gleich, wenn sie die gleiche Struktur besitzen), in Haskells Typsystem, das auf „nominal typing“ basiert (zwei Typen sind gleich, wenn sie den gleichen Namen haben).

Haskells Typen können alternativ durch algebraische Terme höherer Ordnung dargestellt werden [Jones 1995]. Repräsentiert man Typterme durch einen normalen Datentyp, so lassen sich generische Funktionen als Haskell-Funktionen realisieren [Cheney und Hinze 2002, Hinze 2003]. Allerdings ist Haskells **data**-Konstrukt nicht ausdrucksstark genug, um dies zu bewerkstelligen, so dass sich eine Reihe neuer, interessanter Forschungsprobleme ergeben (vergleiche Arbeitspaket **T-2**).

2.2.2 Implementierung

Spezialisierung Wir verfügen über langjährige Erfahrung in der Implementierung generischer Spracherweiterungen. Die aktuelle Version des Generic Haskell-Compilers unterstützt die Definition generischer Funktionen und generischer Typen im M-Stil. Der Compiler ist als Präprozessor realisiert: ein Generic Haskell-Programm wird in ein Haskell-Programm (Haskell 98 erweitert um Rang-2 Polymorphismus) überführt, das mit einem gängigen Haskell-Compiler in ein ausführbares Programm übersetzt werden kann. Eine generische Funktion

wird dabei auf eine typindizierte Familie polymorpher Funktionen abgebildet; die Spezialisierung einer Funktion auf den Typ τ entspricht formal der Interpretation von τ in einem entsprechenden Umgebungsmodell [Hinze 2000c].

Typklassen Wird die generische Sprache eingeschränkt (keine generische Typen, nur sortenindizierte Typen mit einem Argument), so kann auch Haskell 98 (ohne jegliche Erweiterungen) als Zielsprache verwendet werden [Hinze 2004]. Dabei wird Haskells Klassensystem ausgenutzt, um polymorphe Funktionen vom Rang 2 zu simulieren. Dieser Implementierungsansatz eignet sich darüber hinaus dazu, Prinzipien der generischen Programmierung mit Hilfe einer „nicht-generischen“ Sprache zu vermitteln.

2.2.3 Anwendungen

Der Generic Haskell-Compiler hat uns in die Lage versetzt, generische Techniken auch praktisch anzuwenden. Die Palette der Applikationen reicht dabei von grundlegenden Datenstrukturen wie digitalen Suchbäumen [Hinze 2000b] und „Navigationsbäumen“ (Huets „zipper“ Datentyp) [Hinze und Jeuring 2003a], über Memo-Funktionen [Hinze 2000d] bis hin zu XML-Kompressoren [Hinze und Jeuring 2003a]. Die letzte Anwendungen ist besonders interessant; zeigt sie doch, wie Typinformationen in Form von „Document Type Definitions“ genutzt werden können, um gute Kompressionsraten bei vergleichsweise geringem Implementierungsaufwand zu erzielen.

3 Ziele und Arbeitsprogramm

3.1 Ziele

Ziel des beantragten Projektes ist der Entwurf, die Implementierung und die Anwendung einer generischen funktionalen Programmiersprache der zweiten Generation. Wir wollen bisherige, sprachspezifische ad-hoc-Lösungen auf eine allgemeine Basis stellen, ohne dabei die Implementation aus den Augen zu verlieren: Wie soll eine ideale generische Programmiersprache aussehen? Wie kann sie effizient implementiert werden? Welche bekannten Anwendungen lassen sich effizienter oder eleganter lösen und welche neuen Anwendungen erschließen sich?

Generische Programmierung ist ein aufstrebender, wenn auch relativ junger Forschungszweig innerhalb der Praktischen Informatik. Trotz des geringen Alters gibt es bereits eine Vielzahl theoretischer Konzepte und verschiedene Umsetzungen dieser Konzepte. In diesem Projekt soll die zum Teil diverse Theorie zusammengeführt, vereinheitlicht und verallgemeinert werden. Auf Basis einer soliden theoretischen Fundierung wollen wir dann eine generische Programmiersprache in der Tradition der Haskell-Sprachfamilie entwerfen. Die Sprache Haskell dient uns dabei als Ausgangssprache, da Haskell zum einen in Forschung und Lehre weit verbreitet ist und wir zum anderen über reichhaltige Erfahrung sowohl mit Haskell selbst als auch mit Haskell-Übersetzern und anderen Werkzeugen verfügen. Gleichwohl verstehen wir uns nicht als ein Haskell-spezifisches Projekt. Bei den theoretischen Arbeiten wollen wir die Anwendbarkeit auf andere Sprachen nicht aus den Augen verlieren. Wir erwarten, dass die generische Programmierung mit der zunehmenden Verwendung statischer Typsysteme bzw. analoger Konzepte wie zum Beispiel XML-Schemata weiterhin an Bedeutung gewinnt und viele Programmiersprachen um generische Konzepte erweitert werden. Wir erhoffen uns, dass das beantragte Projekt einen wesentlichen Einfluss auf die Ausgestaltung dieser Spracherweiterungen hat.

Neben der soliden theoretischen Fundierung ist für uns die praktische Umsetzung unserer Arbeit von großer Bedeutung. Für das letzte Jahr des Projektes ist daher die Anbindung an einen vollwertigen und weithin genutzten Übersetzer, den Glasgow Haskell Compiler, geplant. Wir sind uns durchaus der Tatsache bewusst, dass dies ein sehr ambitioniertes Ziel ist. Alternativen wie ein separates Werkzeug oder ein Präprozessor bedeuten aber für den generischen Programmierer nicht nur einen zusätzlichen Installations- und Konfigurationsaufwand, sondern erfordern bei Weiterentwicklung des zugrundeliegenden Übersetzers auch einen hohen Anpassungsaufwand. Zudem erschließt sich durch eine direkte Einbindung ein erweiterter Nutzerkreis—viele Programmierer sind eher gewillt, eine zusätzliche Spracherweiterung zu benutzen als von einem separaten Präprozessor abhängig zu sein.

Die entworfene Sprache soll schließlich in verschiedenen Anwendungsstudien eingesetzt und evaluiert werden. Der Schwerpunkt wird dabei nicht auf der Implementierung einzelner, spezieller Beispiele liegen, sondern es ist beabsichtigt, allgemeine Anwendungsgebiete wie zum

Beispiel „Refactoring“ oder XML-Transformationen zu erschließen. Aus diesen Studien wird sich der Bedarf für generische Bibliotheken ableiten, die wir erstellen wollen, um so dem Programmierer den Einstieg in die Welt der generischen Programmierung zu erleichtern.

3.2 Arbeitsprogramm

Das Arbeitsprogramm deckt den gesamten Antragszeitraum von insgesamt drei Jahren ab. Es gliedert sich in drei Arbeitsschwerpunkte, die die Hauptziele des Projektes widerspiegeln.

3.2.1 Theorie und Sprachentwurf

In diesen Arbeitspaketen soll eine generische funktionale Programmiersprache (in der Tradition der Haskell-Sprachfamilie) entworfen und semantisch untermauert werden. Wir verfolgen beim Entwurf zwei konkurrierende Ansätze, die jeweils unterschiedliche Stärken und Schwächen aufweisen, mit dem Ziel, eine ideale Kombination der beiden Ansätze zu finden. Neben dem Sprachentwurf soll ein besonderer Schwerpunkt auf der Programmiermethodik liegen: Wie kann eine generische Funktion spezifiziert werden? Wie kann aus der Spezifikation systematisch ein Programm abgeleitet werden?

Arbeitspaket T-1: Verallgemeinerte generische Funktionen Aktuelle Ansätze zur generischen Programmierung [Clarke et al. 2002, Alimarine und Plasmeijer 2001] unterliegen einer Reihe von Einschränkungen. Ziel dieses Arbeitspaketes ist die Aufhebung oder zumindest Lockerung dieser Einschränkungen.

- Eine generische Funktion kann zum Beispiel nicht an eine andere Funktion (generisch oder nicht-generisch) übergeben werden: generische Funktionen sind wie auch Haskells Typklassen Bürger zweiter Klasse. Generische Funktionen höherer Ordnung sind aber unabdingbar für die Implementierung allgemeiner Traversierungsfunktionen wie sie zum Beispiel beim „Refactoring“ benötigt werden (siehe auch Arbeitspaket **A-1**).
- Gegenwärtig wird eine generische Funktion durch Induktion über die Struktur von Typen definiert. Eine solche Definition ist der Form nach ein *Katamorphismus* (R-Stil und A-Stil). Die Form der Definition bedingt, dass als Typargumente nur flache Muster der Form $C\ \alpha_1\ \ldots\ \alpha_n$ zulässig sind. Wie auch bei der Programmierung nichtgenerischer Funktionen sind jedoch häufig geschachtelte Muster notwendig oder zumindest wünschenswert, etwa für die Behandlung von Spezialfällen (eine generische Ausgabefunktion soll Listen von Zeichen anders formatieren als Listen anderer Werte).

  $$\begin{aligned} \mathit{show}\ \langle [\mathit{Char}] \rangle\ x &= \texttt{"\textbackslash""} +\!\!+ x +\!\!+ \texttt{"\textbackslash""} \\ \mathit{show}\ \langle [\alpha] \rangle\ x &= \texttt{"["} +\!\!+ \mathit{intersperse}\ \texttt{','}\ [\mathit{show}\ \langle \alpha \rangle\ a \mid a \leftarrow x] +\!\!+ \texttt{"]"} \end{aligned}$$

  Kurz gesagt, gilt es, generische Funktionen von Typkatamorphismen zu allgemeinen rekursiven Funktionen auf Typen zu verallgemeinern. Insbesondere sollen auch *mehrparametrige* generische Funktionen ermöglicht werden.

Beide Erweiterungen können prinzipiell durch Rückführung auf die gegenwärtige Kernsprache realisiert werden: Funktionen höherer Ordnung lassen sich durch „firstification“ [Hughes 1996] eliminieren, komplexe Muster können in geschachtelte Fallunterscheidungen übersetzt werden

[Wadler 1987b]. Da bei der „firstification“ die Gefahr der Codeexplosion besteht, streben wir jedoch eine direkte Umsetzung von generischen Funktionen höherer Ordnung an. Zu diesem Zweck wollen wir den interpretativen Ansatz mit dem hierarchischen Ansatz verbinden. Wichtige Erkenntnisse für diese Integration versprechen wir uns insbesondere von den Ergebnissen aus Arbeitspaket **T-2**.

Arbeitspaket T-2: Typrepräsentationen In diesem Arbeitspaket soll ein alternativer Ansatz zur generischen Programmierung untersucht werden, der eine wesentlich engere Kopplung generischer und nicht-generischer Funktionen erlaubt.

Mit der Einführung generischer Funktionen werden viele Konzepte dupliziert: neben „normalen“ Funktionsdefinitionen gibt es generische Definitionen, neben Signaturen gibt es generische Signaturen, neben Mustern gibt es Typmuster usw. Diese Verdopplung von Konzepten lässt sich vermeiden, wenn Typen durch normale Werte repräsentiert werden. Ein in dieser Hinsicht besonders vielversprechender Ansatz basiert auf so genannten *verallgemeinerten Datentypen* („phantom types“ [Cheney und Hinze 2003], induktiven Familien von Typen [Dybjer 1991]). Die grundlegende Idee besteht darin, jedem Typ τ einen Konstruktor des gleichen Namens vom Typ $Rep\ \tau$ zuzuordnen. Der Typ *Int* wird zum Beispiel durch den Konstruktor *Int* vom Typ *Rep Int* dargestellt. Parametrisierte Datentypen werden entsprechend durch parametrisierte Konstruktoren repräsentiert:

$$Tree \quad :: \quad Rep\ \alpha \to Rep\ (Tree\ \alpha)$$

Man beachte, dass *Rep* kein „normaler“ Datentyp ist, da die Konstruktoren nicht den Ergebnistyp $Rep\ \alpha$ besitzen (wie in Haskell und anderen Sprachen erforderlich), sondern *Rep Int* bzw. $Rep\ (Tree\ \alpha)$.

Da Typkonstruktoren durch Wertekonstruktoren repräsentiert werden, sind generische Funktionen „normale“ Funktionen, die ein oder mehrere Argumente vom Typ $Rep\ \alpha$ haben. Damit sind insbesondere keine weiteren Anstrengungen vonnöten, um geschachtelte Typmuster oder generische Funktionen höherer Ordnung zu realisieren (siehe Arbeitspaket **T-1**). Weiterhin können auf einfache Art und Weise *dynamische Werte* realisiert werden und generische Funktionen auf dynamische Werte erweitert werden [Cheney und Hinze 2002, Hinze 2003]. Dafür ergeben sich eine Reihe anderer interessanter Forschungsprobleme.

- Der Repräsentationstyp *Rep* muss ein *offener* Datentyp sein, der jederzeit um weitere Konstruktoren erweitert werden kann (immer dann, wenn ein neuer Datentyp eingeführt wird). Die meisten gängigen Sprachen erlauben ausschließlich *geschlossene* Datentypen, bei denen die Konstruktoren zum Definitionszeitpunkt vollständig aufgezählt werden. Es muss daher untersucht werden, ob *Rep* als eingebauter, spezieller Datentyp zur Sprache hinzugefügt werden kann, oder ob allgemein die Fähigkeit zur Definition offener Datentypen erstrebenswert ist. Ein ähnliches Problem ergibt sich für die generischen Funktionen selbst, die man um neue Fälle erweitern möchte, falls für neu eingeführte Datentypen ein spezielles (nicht das generisch abgeleitete) Verhalten erwünscht ist.
- Mittels Fallunterscheidung auf Typrepräsentationen definierte generische Funktionen unterscheiden Typen anhand ihres Namens und nicht ihrer Struktur („nominal typing“ versus „structural typing“). Die eigentliche Stärke generischer Funktionen besteht aber

gerade darin, die Struktur eines Typs auszunutzen, um Codevervielfachung zu vermeiden. Mit anderen Worten: ein Repräsentationstyp löst zwar das Problem der Analyse von Datentypen, hilft aber nicht bei der Transformation von Datentypen in eine geeignete strukturelle Repräsentation. Beide Teilaspekte müssen auf geeignete Weise miteinander kombiniert werden.

- Wählt man einen Datentyp zur Repräsentation von Typargumenten, wird die *Anwendung* generischer Funktionen umständlich: die Typrepräsentation muss beim Aufruf explizit angegeben werden, obwohl sich die Repräsentation unter Umständen eindeutig rekonstruieren lässt (die automatische Rekonstruktion von Typen ist gerade ein großer Vorteil von Haskells Klassensystem). Da Typen durch Werte repräsentiert werden, ist statt Typinferenz „Codeinferenz“ wünschenswert, um die Typrepräsentation automatisch ergänzen zu können, sofern sie aus dem Kontext eindeutig bestimmt werden kann.

Arbeitspaket T-3: Sichten Eine Sicht (engl. view) [Wadler 1987a, Okasaki 1998] erlaubt es, einen Typ, insbesondere einen abstrakten Typ, als *freien* Datentyp zu behandeln (zum Beispiel einen Binomialbaum als Liste). Sichten werden implizit bei der Implementierung generischer Funktionen verwendet: jedem Typ wird ein *Strukturtyp* (eine Summe von Produkten) zugeordnet, auf dem die generischen Funktionen arbeiten. Ziel dieses Arbeitspaketes ist die implizite Verwendung explizit zu machen: es soll ermöglicht werden, den Strukturtyp für jeden Datentyp selbst zu bestimmen. Weiterhin ist daran gedacht, pro Typ mehrere Strukturtypen zuzulassen, die verschiedene Sichten auf diesen Typ realisieren. Für die Definition von *Kata-* und *Anamorphismen* [Meijer et al. 1991] muss zum Beispiel die *rekursive* Struktur eines Typs modelliert werden:

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Base}\ \langle \mathit{Fix}\ \phi \rangle\ \alpha & = & \phi\ \alpha \to \alpha \\
\mathit{cata}\ \langle \tau :: \star \rangle & :: & \mathit{Base}\ \langle \tau \rangle\ \alpha \to (\tau \to \alpha) \\
\mathit{cata}\ \langle \mathit{Fix}\ \phi \rangle\ \mathit{alg} & = & \mathit{alg} \cdot \mathit{map}\ \langle \phi \rangle\ (\mathit{cata}\ \langle \mathit{Fix}\ \phi \rangle\ \mathit{alg}) \cdot \mathit{out}
\end{array}
$$

In diesem Beispiel wird der Typ τ als Fixpunkt des Endofunktors ϕ aufgefasst. Erlaubt man pro Typ mehrere Sichten, so muss geklärt werden, wie verschiedene generische Funktionen, die unter Umständen unterschiedliche Sichten verwenden, interagieren.

Schließlich sind Sichten unabdingbar für die generische Behandlung abstrakter Datentypen. Generische Funktionen benötigen die Struktur eines Typs; im Fall abstrakter Datentypen bleibt diese Struktur aber gerade verborgen. Sichten vermitteln zwischen den beiden Extremen, so dass wir generische Funktionen um die transparente Anwendung von Sichten erweitern möchten.

Arbeitspaket T-4: Spezifikation generischer Funktionen In diesem Arbeitspaket soll untersucht werden, wie eine generische Funktion spezifiziert werden kann und wie aus der Spezifikation systematisch ein Programm abgeleitet werden kann. Überraschenderweise ist dieser Themenkreis bisher kaum untersucht worden—die wenigen, uns bekannten Arbeiten sind unten aufgeführt. Dabei sind generische Beweise und Herleitungen von ähnlicher Ökonomie wie generische Funktionen und Typen: ein einzelner Beweis garantiert die Korrektheit einer ganzen Klasse von Funktionen.

Eigenschaften generischer Funktionen können mit Hilfe so genannter *logischer Relationen* (engl. logical relations) beschrieben werden (in der Habilitationsschrift des Antragstellers untersucht). Dieser Ansatz ist allerdings auf generische Funktionen eingeschränkt, die die formale Form eines Typkatamorphismus haben. Offen ist, ob und wie sich die Methode auf verallgemeinerte generische Funktionen (siehe Arbeitspaket **T-1**) erweitern lässt.

Einige generische Funktionen wie zum Beispiel *map* sind allein durch den generischen Typ eindeutig festgelegt. Andere Funktionen wie zum Beispiel der „Kommutator“

$$zip\ \langle \phi :: \star \to \star, \psi :: \star \to \star \rangle \quad :: \quad \phi\ (\psi\ \alpha) \to \psi\ (\phi\ \alpha)$$

der eine ϕ-Struktur von ψ-Strukturen in eine ψ-Struktur von ϕ-Strukturen überführt (ϕ und ψ kommutieren), können durch sogenannte *naturality conditions* eindeutig bestimmen werden [Hoogendijk und Backhouse 1997, Backhouse und Hoogendijk 2003]. Lassen sich diese Einzelbeispiele auf eine interessante Klasse generischer Funktionen verallgemeinern?

3.2.2 Implementierung

In diesen Arbeitspaketen soll die entwickelte generische Programmiersprache implementiert werden. Wie auch beim Sprachentwurf werden für die verschiedenen Implementierungsaspekte konkurrierende Ansätze untersucht und miteinander verglichen. Design und Implementierung werden zeitlich versetzt miteinander verzahnt, gegebenenfalls auch iteriert. Der endgültige Sprachentwurf wird abschließend in den Glasgow Haskell Compiler integriert.

Arbeitspaket I-1: Dependency-style Generic Haskell „Dependency-style“ Generic Haskell (A-Stil) [Löh et al. 2003] ermöglicht es schon jetzt, generische Programme in einer bislang ungekannten Ausdruckskraft zu erstellen. Die in der Doktorarbeit von Andres Löh [Löh 2004] beschriebenen Ansätze sind bislang allerdings noch nicht implementiert.

Wir planen, den existierenden Generic Haskell-Compiler [Clarke et al. 2002] entsprechend zu erweitern. Dazu muss eine neue „Ebene“ in den Übersetzer integriert werden: die Syntax muss an den A-Stil angepasst werden, ein zusätzlicher Übersetzungsschritt führt dann die Programme im A-Stil samt Abhängigkeiten auf Funktionen im derzeit implementierten M-Stil zurück. Zusätzlich muss die Fehlerbehandlung angepasst werden. Für die Unterstützung von generischen Typen ist mehr Arbeit nötig, da diese gegenwärtig nur in eingeschränkter Form implementiert sind. Insbesondere lassen sich keine generischen Typsynonyme definieren (dem Schlüsselwort **type** in Haskell entsprechend), sondern nur generische Datentypen (**data**). Hier gilt es, zunächst diese zusätzliche Funktionalität bereitzustellen und dann in ähnlicher Weise wie bei den generischen Funktionen eine Übersetzung von A- nach M-Stil zu implementieren.

Wir erhoffen uns von der Implementierung neue Erkenntnisse über noch ausstehende Probleme und Inspiration zu neuen Anwendungen. Es besteht darüber hinaus die Erwartung, dass andere Forscher (wir haben mit einer Reihe von Wissenschaftlern eine konkrete Zusammenarbeit vereinbart, siehe Abschnitt 5.2) eine aktuelle Implementierung von Generic Haskell für ihre Projekte verwenden und sich durch das Feedback weitere interessante Forschungsfragen eröffnen.

Arbeitspaket I-2: Implementierung von „typecase“ In diesem Arbeitspaket sollen verschiedene, konkurrierende Implementierungstechniken realisiert und evaluiert werden.

Die gängigste Implementierungsmethode basiert auf der *Spezialisierung* generischer Funktionen: jede generische Funktion wird in eine Familie polymorpher Funktionen übersetzt. Diese Technik wird sowohl im Generic Haskell-Compiler und im Generic Clean-Compiler als auch in vereinfachter Form für Haskells **deriving**-Konstrukt verwendet. Da pro Typ eine polymorphe Instanz der generischen Funktion erzeugt wird, ist potenziell die Gefahr der Codeexplosion gegeben. Erste praktische Erfahrungen mit Generic Clean haben weiterhin gezeigt, dass die Codequalität im Vergleich zu handgeschriebenen Code abfällt. Dies liegt zum einen an der massiven Verwendung von polymorphen Funktionen höherer Ordnung und zum anderen an der Transformation von Typen in Strukturtypen (siehe auch Arbeitspaket **I-3**). Wir hoffen, die Codequalität durch symbolische Auswertung zur Übersetzungszeit verbessern zu können.

Alternativ lässt sich der unter **T-2** beschriebene Ansatz für die Implementierung adaptieren: Typen werden durch spezielle Werte repräsentiert, die zur Laufzeit von den generischen Funktionen analysiert werden. Dieser Ansatz vermeidet die Codeexplosion, ist aber auf Grund des interpretativen Overheads ineffizienter.

Da Typargumente zur Übersetzungszeit statisch bekannt sind, läßt sich der Spezialisierungsansatz als *partielle Auswertung* des interpretativen Ansatzes deuten. Vor diesem Hintergrund wollen wir untersuchen, wie sich beide Implementierungstechniken intelligent kombinieren lassen und wie die Techniken mit der separaten Übersetzung von Modulen in Einklang gebracht werden können. Zum Beispiel ist denkbar, dass lediglich generische Funktionen erster Ordnung spezialisiert werden und bei Funktionen höherer Ordnung auf den interpretativen Ansatz zurückgegriffen wird. Zusätzlich wollen wir dem Benutzer die Möglichkeit an die Hand geben, den Spezialisierungsprozess durch sogenannte *Pragmas* zu steuern (analog zu Haskells `SPECIALIZE` Pragma). Da die Spezialisierung auf dem Programmcode der generischen Funktionen arbeitet, muss für die separate Übersetzung von Modulen zusätzlicher Aufwand getrieben werden (indem zum Beispiel der Quellcode generischer Funktionen in den „interface files“ von Modulen ablegt wird).

Arbeitspaket I-3: Implementierung von Sichten Das unter **T-4** beschriebene Konzept der Sichten soll in diesem Arbeitspaket umgesetzt werden. In Abhängigkeit von den erzielten Ergebnissen werden entweder häufig benötigte generische Sichten „fest verdrahtet“ oder es wird eine eigene Teilsprache für Sichten implementiert.

Weiterhin wollen wir in diesem Arbeitspaket der Frage nachgehen, wie der Overhead, der durch die Transformation von Typen in zugehörige Strukturtypen entsteht, durch „Deforestation“ [Wadler 1988] eliminiert werden kann. Da Strukturtypen *binäre* Summen verwenden, ist dieser Mehraufwand insbesondere bei Datentypen mit einer Vielzahl von Konstruktoren erheblich. Die Spezialisierung der Gleichheit auf den Listentyp illustriert das Problem (der

Code ist gegenüber dem automatisch erzeugten etwas vereinfacht).

$$
\begin{array}{lcl}
equal\,\langle List\rangle\ eqa\ as_1\ as_2 & = & equal\,\langle + \rangle\ (equal\,\langle 1\rangle)\ (equal\,\langle \times\rangle\ eqa\ (equal\,\langle List\rangle\ eqa)) \\
 & & \quad (fromList\ as_1)\ (fromList\ a_2) \\
fromList & :: & [\alpha] \to (1 + \alpha \times [\alpha]) \\
fromList\ [\,] & = & Inl\ () \\
fromList\ (a : as) & = & Inr\ (a, as)
\end{array}
$$

Die Listen as_1 und as_2 werden durch *fromList* in Elemente des zugehörigen Strukturtyps $1 + \alpha \times [\alpha]$ überführt. Der Ausdruck $equal\,\langle + \rangle\ (equal\,\langle 1\rangle)\ (equal\,\langle \times\rangle\ eqa\ (equal\,\langle List\rangle\ eqa))$, der den Strukturtyp auf Funktionsebene widerspiegelt, implementiert den eigentlichen Test auf Gleichheit. In diesem Fall können die Konvertierung in den Strukturtyp und die geschachtelten Funktionsaufrufe—$equal\,\langle + \rangle$ und $equal\,\langle \times\rangle$ sind zudem Funktionen höherer Ordnung—durch lokale Programmtransformationen eliminiert werden (mittels „**case** floating“ und „**case** reduction“, siehe [Peyton Jones 1996]). Der optimierte Funkion gleicht handgeschriebenem Code (vergleiche mit der Klasseninstanz aus Abbschnitt 2.1.1):

$$
\begin{array}{lcl}
equal\,\langle List\rangle\ eqa\ [\,]\ [\,] & = & True \\
equal\,\langle List\rangle\ eqa\ [\,]\ (a_2 : as_2) & = & False \\
equal\,\langle List\rangle\ eqa\ (a_1 : as_1)\ [\,] & = & False \\
equal\,\langle List\rangle\ eqa\ (a_1 : as_2)\ (a_2 : as_2) & = & (eqa\ a_1\ a_2) \wedge (equal\,\langle List\rangle\ eqa\ as_1\ as_2)
\end{array}
$$

Wenn der Strukturtyp Teil einer komplexen Datenstruktur ist, oder gar Argument eines abstrakten Typs, wie zum Beispiel bei der monadischen Funktion $mapM\,\langle \phi :: \star \to \star \rangle :: (Monad\ m) \Rightarrow (\alpha \to m\ \beta) \to (\alpha \to m\ (\phi\ \beta))$, dann sind die Optimierungen nicht mehr so leicht durchzuführen. Für die Optimierungsschritte werden anwendungsspezifische Eigenschaften, wie zum Beispiel die Monadengesetze, benötigt.

Arbeitspaket I-4: Anbindung an den Glasgow Haskell Compiler Der Glasgow Haskell Compiler (GHC) ist der ausgereifteste Übersetzer für die Programmiersprache Haskell. Es ist geplant, die entwickelte generische Sprache in den GHC zu integrieren. Der GHC bietet sich für die Anbindung in besonderer Weise an, da die im Übersetzer verwendete Zwischensprache (ein polymorpher λ-Kalkül mit algebraischen Datentypen, „external core“) für andere „Frontends“ zugänglich ist. Auf diese Weise kann die gesamte Infrastruktur des „Backends“ wiederverwendet werden. Darüber hinaus kann so von generischen Programmen auf die umfangreiche Haskell-Bibliothek zurückgegriffen werden bzw. können generische Programme mit Haskell-Programmen kombiniert werden.

In Abhängigkeit vom Design der generischen Sprache müssen eine Vielzahl von Teilaufgaben angegangen werden:

- Für das neu zu erstellende Frontend muss ein Parser und ein Typechecker implementiert werden. Für die Syntaxanalyse lassen sich entsprechende Teile des Generic Haskell-Compilers wiederverwenden (über einen Typechecker verfügt der Compiler nicht). Alternativ kann das Haskell-Frontend des GHC entsprechend modifiziert werden. Zum gegenwärtigen Zeitpunkt lässt sich nicht abschätzen, welche Alternative mit weniger Implementierungsaufwand verbunden ist.

- Je komplexer eine Programmiersprache ist, desto wichtiger (und auch schwieriger) wird es, im Fehlerfall aussagekräftige Fehlermeldungen zu generieren. Da generische Programmiersprachen ihrer Natur nach einen hohen Abstraktionsgrad aufweisen, wollen wir diesem Punkt besondere Beachtung schenken.
- Wenn verallgemeinerte Datentypen in die generische Sprache aufgenommen werden (siehe Arbeitspaket **T-2**), so muss die Zwischensprache („external core“) entsprechend erweitert werden—die eigentliche Codeerzeugung im GHC bleibt von der Erweiterung jedoch unberührt.
- Die in den Arbeitspaketen **I-2** und **I-3** entwickelten Optimierungen müssen umgesetzt werden. Da es sich um sprachspezifische Optimierungen handelt, die generische, nicht aber „normale“ Funktionen betreffen, planen wir, die entsprechenden Transformationen in das Frontend zu integrieren. Unter Umständen kann es sich jedoch als günstiger erweisen, die Optimierungen in das Backend auszulagern, etwa wenn die sprachspezifischen Optimierungen durch andere Transformationen (zum Beispiel „inlining“) erst ermöglicht werden.

3.2.3 Anwendungen

In diesen Arbeitspaketen wird die entworfene generische Programmiersprache in verschiedenen Anwendungsstudien evaluiert (gegebenenfalls wiederholt). Bekannte Anwendungen wie zum Beispiel Kompression von XML-Dokumenten [Hinze und Jeuring 2003a] oder generische Datenkonversion [Jansson und Jeuring 2002] werden daraufhin untersucht, ob sie sich eleganter oder effizienter (in Bezug auf die Entwicklungszeit) lösen lassen. Darüber hinaus wird angestrebt, neue Anwendungsfelder für die generische Programmierung zu erschließen: zum Beispiel *typsichere* Anbindungen an relationale Datenbanken.

Arbeitspaket A-1: Generisches Refactoring „Refactorings“ sind Programmtransformationen auf Quellebene („source to source“), die die Struktur und Organisation eines Programms ändern, nicht aber dessen Funktionalität. Derartige Transformationen lassen sich in natürlicher Weise durch generische Funktionen implementieren, die auf der abstrakten Syntax einer Programmiersprache arbeiten. „Refactorings“ sind typischerweise komplexe Kombinationen elementarer Transformationen, so dass an dieser Stelle generische Funktionen höherer Ordnung benötigt werden. Darüber hinaus muss man typ- bzw. konstruktorspezifisches Verhalten programmieren können: ein „Refactoring“ lässt sich oft als Variante einer generischen Funktion formulieren, die sich für einen speziellen Datentyp oder einen speziellen Konstruktor anders verhält. Ein einfaches Beispiel ist die Bestimmung der freien Variablen eines Programmfragmentes: die zugrundeliegende generische Funktion entspricht einem einfachen Baumdurchlauf, der *alle* Variablen bestimmt; konstruktorspezifisches Verhalten ist an den Bindungsstellen nötig: dort müssen die gebundenen Variablen aus der Menge der freien Variablen entfernt werden.

In diesem Arbeitspaket soll eine Bibliothek grundlegender „Refactorings“ nach dem Vorbild des Haskell Refactorers [Li et al. 2003] erstellt werden. Die aktuelle Implementierung des Haskell Refactorers greift neben handgeschriebenem Code auf Techniken der kombinatorbasierten generischen Programmierung zurück, so dass die Möglichkeit gegeben ist, unsere Implementierung mit dem vorhandenen Code entlang verschiedener Software-Metriken zu vergleichen.

Arbeitspaket A-2: Automatisches Testen Das Testen von Programmen ist ein wesentlicher Bestandteil des Softwareentwicklungsprozesses. Beim automatischen Testen übernehmen Programme die systematische Generierung von Testdaten. Abhängig von der Komplexität der Daten kann die Implementierung der Testprogramme aufwändiger sein als die Implementierung der zu testenden Programme. Da Datengenerierung der Natur nach ein generisches Problem ist, können auch für diese Anwendung sinnvoll Methoden der generischen Programmierung verwendet werden. Die folgende einfache Generierungsfunktion mag als Illustration dienen: zu einem gegebenen Typ erzeugt *enum* einen (möglicherweise unendlichen) Binärbaum, der *alle* Werte des Typs enthält.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Bush}\ \alpha & = & \mathit{Leaf}\ \alpha \mid \mathit{Fork}\ (\mathit{Bush}\ \alpha)\ (\mathit{Bush}\ \alpha) \\
\mathit{enum}\ \langle \tau :: \star \rangle & :: & \mathit{Bush}\ \tau \\
\mathit{enum}\ \langle 1 \rangle & = & \mathit{Leaf}\ () \\
\mathit{enum}\ \langle \alpha + \beta \rangle & = & \mathit{Fork}\ (\mathit{map}\ \langle \mathit{Bush} \rangle\ \mathit{Inl}\ (\mathit{enum}\ \langle \alpha \rangle))\ (\mathit{map}\ \langle \mathit{Bush} \rangle\ \mathit{Inr}\ (\mathit{enum}\ \langle \beta \rangle)) \\
\mathit{enum}\ \langle \alpha \times \beta \rangle & = & \mathit{bind}\ (\mathit{enum}\ \langle \alpha \rangle)\ (\lambda a \to \mathit{map}\ \langle \mathit{Bush} \rangle\ (\lambda b \to (a, b))\ (\mathit{enum}\ \langle \beta \rangle)) \\
\mathit{bind} & :: & \mathit{Bush}\ \alpha \to (\alpha \to \mathit{Bush}\ \beta) \to \mathit{Bush}\ \beta \\
\mathit{bind}\ (\mathit{Leaf}\ a)\ k & = & k\ a \\
\mathit{bind}\ (\mathit{Fork}\ l\ r)\ k & = & \mathit{Fork}\ (\mathit{bind}\ l\ k)\ (\mathit{bind}\ r\ k)
\end{array}
$$

Das generische Programm ist insofern bemerkenswert, als es Daten erzeugt (nicht konsumiert oder transformiert)—nicht alle generischen Ansätze unterstützen dies.

In diesem Arbeitspaket sollen verschiedene generische Generierungsfunktionen entwickelt und in das populäre Testwerkzeug „QuickCheck“ integriert werden [Claessen und Hughes 2000, Claessen und Hughes 2002]. QuickCheck ist eine Haskell-Bibliothek, die den Programmierer bei der Formulierung und Durchführung von Programmtests unterstützt. Die Bibliothek bietet allerdings nur sehr elementare Möglichkeiten, systematisch Testdaten zu erzeugen. Die vorhandene Grundfunktionalität soll um fortgeschrittene Generierungsfunktionen erweitert werden, die zum Beispiel nur einen Ausschnitt des Wertebereichs erzeugen und dabei Häufigkeitsverteilungen oder anwendungsspezifische Anforderungen an die Testdaten (Erzeugung von Rand- oder Extremwerten) berücksichtigen.

Arbeitspaket A-3: XML und Datenkonversion Die „Extensible Markup Language“ (XML) [Bray et al. 2004] hat sich in den letzten Jahren zum de-facto Standard für die Beschreibung von Dokumenten bzw. allgemein von Daten entwickelt. Der „Markup“, den ein XML Dokument verwendet, ist in der so genannten „Document Type Definition“ (DTD) festgelegt. Ähnlich wie Typen die Struktur von Werten festlegen, beschreiben DTDs die Struktur von XML-Daten. Verarbeitung von XML-Dokumenten ist daher ein naheliegendes Anwendungsgebiet der generischen Programmierung.

Da XML-Dokumente hierarchisch strukturiert sind—im Prinzip haben wir es mit annotierten Mehrwegbäumen zu tun—lassen sich viele Operationen auf XML-Daten in natürlicher Weise *funktional* beschreiben. In der Tat sind viele Spezialsprachen, die im Zusammenhang mit XML entwickelt worden sind, wie etwa XSL Transformations (XSLT) [Clark 1999] oder XDUCE [Hosoya und Pierce 2003] im Kern funktionale Programmiersprachen. In diesem Arbeitspaket soll eine einfache Bibliothek entwickelt werden, die den Benutzer bei der Generierung

und Transformation von XML-Dokumenten unterstützt. Dabei wollen wir uns in Umfang und Funktionalität an der Haskell-Bibliothek HaXml [Wallace und Runciman 1999] orientieren, die auf einer „ungetypten“ Repräsentation von XML basiert (DTDs werden nicht berücksichtigt).

3.2.4 Projektablaufplan

Der unten abgebildete Projektablaufplan gibt einen Überblick über die ungefähre zeitliche Abfolge der Arbeitspakete.

Jahr 1	Jahr 2		Jahr 3	
T-1	**A-1**	**T-3**	**T-4**	
T-2	**I-2**		**I-4**	
I-1	**I-3**	**A-2**	**A-3**	**I-4**

Da von den Anwendungsstudien wichtige Anregungen für den Sprachentwurf zu erwarten sind, ist geplant, die Arbeitspakete **A-1** bis **A-3** wann immer möglich zeitlich vorzuziehen (gegebenenfalls in Teilen).

Wir erwarten, dass die Integration der Spracherweiterung in den GHC (Arbeitspaket **I-4**) am Ende des Projektzeitraums einen Großteil der Kapazität in Anspruch nehmen wird. Daher ist am Ende von Jahr 3 vorgesehen, nur zwei statt wie sonst drei Arbeitspakete parallel zu bearbeiten.

3.3 Untersuchungen am Menschen

Nicht gegeben.

3.4 Tierversuche

Nicht gegeben.

3.5 Gentechnologische Experimente

Nicht gegeben.

4 Beantragte Mittel

4.1 Personalkosten

1 wissenschaftliche Mitarbeiterstelle BAT IIa (Informatiker) für 2 + 1 Jahre.

Es ist vorgesehen, die Stelle mit Andres Löh zu besetzen, der sich in seiner Dissertation intensiv mit Erweiterungen von Generic Haskell und alternativen Ansätzen zur generischen Programmierung beschäftigt hat und auf Grund dessen fachlich hochqualifiziert ist. Herr Löh arbeitet zur Zeit an der Universität Utrecht und wird voraussichtlich am 2. September 2004 promovieren.

1 studentische Hilfskraft à 19 Stunden pro Woche für 2 + 1 Jahre.

Die Hilfskraftstelle wird für die umfangreichen Implementierungsarbeiten und für die Durchführung der Anwendungsstudien benötigt. Die Arbeit ist sehr anspruchsvoll, da fundierte Kenntnisse aus verschiedenen Gebieten der Informatik benötigt werden (Übersetzerbau, Typsysteme und Typinferenzsysteme, XML etc).

4.2 Wissenschaftliche Geräte

Für den Erfolg des Projektes ist es wichtig, Lösungsansätze und geleistete Implementierungsarbeiten frühzeitig international zu präsentieren. Um insbesondere die erstellte Software auf Fachtagungen vorführen zu können, möchten wir einen Notebook Computer beantragen (Notebook Computer gehören nicht zur Grundausstattung des Instituts).

Sony Vaio Notebook (VGN-S1XP oder vergleichbar)	2500 €
Anschaffungskosten 4.2	2500 €

4.3 Verbrauchsmaterial

Keine.

4.4 Reisen

Für die Zusammenarbeit mit den unter Abschnitt 5.2 genannten Wissenschaftlern und für die Teilnahme an jährlich drei internationalen Konferenzen bzw. Arbeitstreffen von IFIP Working Groups beantragen wir jährlich 8500€. Die Möglichkeit, Ergebnisse frühzeitig vorzustellen und Forschungsfragen mit den international führenden Wissenschaftlern auf diesem Gebiet zu diskutieren, ist ein wichtiger Faktor für den Erfolg des Projektes.

1. Jahr:	Zusammenarbeit (siehe Abschnitt 5.2)	1000€
	Kongressreise (2×1250€ =)	2500€
	Kongressreise (2×1250€ =)	2500€
	Kongressreise (2×1250€ =)	2500€
2. Jahr:	ditto	8500€
3. Jahr:	ditto	8500€
Summe 4.4		25.500€

Wir planen Arbeiten auf folgenden Kongressen einzureichen bzw. an folgenden Arbeitstreffen teilzunehmen:

- ACM Symposium on Principles of Programming Languages (POPL),
- ACM Conference on Programming Language Design and Implementation (PLDI),
- International Conference on Functional Programming (ICFP),
- International Conference on Mathematics of Program Construction (MPC),
- European Symposium on Programming (ESOP),
- ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),
- European Conference on Object-Oriented Programming (ECOOP),
- IFIP Working Group 2.1 (Algorithmic Languages and Calculi),
- IFIP Working Group 2.8 (Functional Programming).

4.5 Publikationskosten

Keine.

4.6 Sonstige Kosten

Keine.

5 Voraussetzungen für die Durchführung des Vorhabens

5.1 Zusammensetzung der Arbeitsgruppe

Hochschuldozent Dr. Ralf Hinze (Antragsteller, circa 20%)
Institut für Informatik III
Universität Bonn

Der Projektantrag dient dem Aufbau einer eigenen Arbeitsgruppe. Der Antragsteller wird das Projekt im gesamten Förderungszeitraum intensiv wissenschaftlich begleiten.

Zusätzlich werden im Rahmen des Projektes Diplomarbeitsthemen vergeben.

5.2 Zusammenarbeit mit anderen Wissenschaftlern

Mit folgenden Wissenschaftlern ist eine konkrete Zusammenarbeit vereinbart:

- Prof. Dr. Roland Backhouse (University of Nottingham)—Spezifikation und Herleitung generischer Programme,
  Prof. Dr. Richard Bird (Oxford University)—Spezifikation und Herleitung generischer Programme,
- Prof. Dr. Armin B. Cremers (Universität Bonn)—Datenbanken und XML,
- Prof. Dr. Johan Jeuring (Universiteit Utrecht, Open Universiteit Nederland)—Theorie und Anwendung der generischen Programmierung,
- Prof. Dr. Rainer Manthey (Universität Bonn)—Datenbanken,
- Prof. Simon Peyton Jones (Microsoft Research Cambridge)—Glasgow Haskell Compiler,
- Prof. Dr. ir. M.J. Plasmeijer (Katholieke Universiteit Nijmegen)—Clean Compiler,
- Prof. Dr. Simon J. Thompson (University of Kent)—Refactoring.

Die Zusammenarbeit ist keine notwendige Voraussetzung für den Erfolg des Projektes.

5.3 Arbeiten im Ausland und Kooperation mit ausländischen Partnern

Keine.

5.4 Apparative Ausstattung

Die apparative Ausstattung des Instituts für Informatik III besteht aus einem lokalen Netz von SUN- und PC-Arbeitsplätzen, die den wissenschaftlichen Mitarbeitern zur Verfügung stehen.

5.5 Laufende Mittel für Sachausgaben

Druckkosten, Telefonkosten sowie weitere Verbrauchskosten werden durch die Grundausstattung des Instituts getragen.

5.6 Sonstige Voraussetzungen

Keine.

6 Erklärungen

6.1

Ein Antrag auf Finanzierung dieses Vorhabens wurde bei keiner anderen Stelle eingereicht. Wenn ich einen solchen Antrag stelle, werde ich die Deutsche Forschungsgemeinschaft unverzüglich benachrichtigen.

6.2

Der Vertrauensdozent, Prof. Dr. Michael Famulok, ist von der Antragstellung unterrichtet worden.

6.3

Es besteht keine Zugehörigkeit zu einem Max-Planck–Institut.

7 Unterschrift

(Hochschuldozent Dr. Ralf Hinze)

8 Verzeichnis der Anlagen

- Fragebogen für Antragsteller,
- Curriculum Vitæ (inklusive Veröffentlichungsliste des Antragstellers),
- Kopie „A New Approach to Generic Functional Programming“ (POPL'00),
- Offprint „Polytypic values possess polykinded types“ (SCP 2002),
- Kopie „Dependency-style Generic Haskell“ (ICFP 2003),
- Offprint „Type-indexed data types“ (SCP 2004),
- elektronische Kopie des Antrages auf CD-ROM.

Die beigefügten Unterlagen können nach Abschluss der Begutachtung bei der DFG verbleiben.

Literaturverzeichnis

[Achten et al. 2004] Achten, Peter, M. van Eekelen und R. Plasmeijer (2004). *Generic Graphical User Interfaces*. In: *Implementation of Functional Languages: 15th International Workshop, IFL 2003, Edinburg, Scotland, September 8–10, 2003, Revised Selected Papers*, Lecture Notes in Computer Science. Springer-Verlag. To appear.

[Achten und Hinze 2002] Achten, Peter und R. Hinze (2002). *Combining Generics and Dynamics*. Technischer Bericht NIII-R0206, Nijmegen Institute for Computing and Information Sciences, University of Nijmegen.

[Alimarine und Plasmeijer 2001] Alimarine, Artem und R. Plasmeijer (2001). *A Generic Programming Extension for Clean*. In: Arts, Th. und M. Mohnen, Hrsg.: *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers, Älvsjö, Sweden*, Bd. 2312 d. Reihe *Lecture Notes in Computer Science*, S. 168–185. Springer-Verlag.

[Alimarine und Smetsers 2004] Alimarine, Artem und S. Smetsers (2004). *Optimizing Generic Functions*. In: *Proceedings of the Seventh International Conference on Mathematics of Program Construction, Stirling, United Kingdom*. To appear.

[Backhouse und Hoogendijk 2003] Backhouse, Roland und P. Hoogendijk (2003). *Generic Properties of Datatypes*. In: Backhouse, Roland und J. Gibbons, Hrsg.: *Generic Programming: Advanced Lectures*, Bd. 2793 d. Reihe *Lecture Notes in Computer Science*, S. 97–132. Springer-Verlag.

[Bird und de Moor 1997] Bird, Richard und O. de Moor (1997). *Algebra of Programming*. Prentice Hall Europe, London.

[Bird und Meertens 1998] Bird, Richard und L. Meertens (1998). *Nested Datatypes*. In: Jeuring, J., Hrsg.: *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, Bd. 1422 d. Reihe *Lecture Notes in Computer Science*, S. 52–67. Springer-Verlag.

[Bray et al. 2004] Bray, Tim, J. Paoli, C. Sperberg-McQueen, E. Maler, F. Yergeau und J. Cowan (2004). *Extensible Markup Language (XML) 1.1*. Technischer Bericht `http://www.w3.org/TR/xml11`, World Wide Web Consortium.

[Cheney und Hinze 2002] Cheney, James und R. Hinze (2002). *A Lightweight Implementation of Generics and Dynamics*. In: Chakravarty, Manuel M.T., Hrsg.: *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, S. 90–104. ACM Press.

[Cheney und Hinze 2003] Cheney, James und R. Hinze (2003). *First-Class Phantom Types*. Technischer Bericht, Cornell University.

[Claessen und Hughes 2000] Claessen, Koen und J. Hughes (2000). *QuickCheck: a lightweight tool for random testing of Haskell programs*. ACM SIGPLAN Notices, 35(9):268–279.

[Claessen und Hughes 2002] Claessen, Koen und J. Hughes (2002). *Testing monadic code with QuickCheck*. ACM SIGPLAN Notices, 37(12):47–59.

[Clark 1999] Clark, James (1999). Hrsg.: *XSL Transformations (XSLT) Version 1.0*. Technischer Bericht `http://www.w3.org/TR/xslt`, World Wide Web Consortium.

[Clarke et al. 2001] Clarke, Dave, R. Hinze, J. Jeuring, A. Löh und J. de Wit (2001). *The Generic Haskell User's Guide, Version 0.99 (Amber)*. Technischer Bericht UU-CS-2001-26, Institute of Information and Computing Sciences, Universiteit Utrecht.

[Clarke et al. 2002] Clarke, Dave, J. Jeuring und A. Löh (2002). *The Generic Haskell User's Guide, Version 1.23 – Beryl release*. Technischer Bericht UU-CS-2002-047, Institute of Information and Computing Sciences, Universiteit Utrecht.

[Clarke und Löh 2003] Clarke, Dave und A. Löh (2003). *Generic Haskell, Specifically*. In: Gibbons, Jeremy und J. Jeuring, Hrsg.: *Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming July 11–12, 2002, Dagstuhl, Germany*, Nr. 115 in *International Federation for Information Processing*, S. 21–47. Kluwer Academic Publishers.

[Cockett und Fukushima 1992] Cockett, Robin und T. Fukushima (1992). *About Charity*. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary.

[Dubois et al. 1995] Dubois, Catherine, F. Rouaix und P. Weis (1995). *Extensional polymorphism*. In: ACM, Hrsg.: *Conference record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 22–25, 1995*, S. 118–129, New York, NY, USA. ACM Press.

[Dybjer 1991] Dybjer, Peter (1991). *Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics*. In: Huet, Gerard und G. Plotkin, Hrsg.: *Logical Frameworks*, S. 280–306. Prentice Hall.

[Garcia et al. 2003] Garcia, Ronald, J. Jarvi, A. Lumsdaine, J. Siek und J. Willcock (2003). *A comparative study of language support for generic programming*. ACM SIGPLAN Notices, 38(11):115–134.

[Gibbons 2003] Gibbons, Jeremy (2003). *Patterns in Datatype-Generic Programming*. In: *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages. Uppsala, 25th August 2003*.

[Hall et al. 1996] Hall, Cordelia V., K. Hammond, S. L. Peyton Jones und P. L. Wadler (1996). *Type classes in Haskell*. ACM Transactions on Programming Languages and Systems, 18(2):109–138.

[Harper und Morrisett 1995a] Harper, Robert und G. Morrisett (1995a). *Compiling Polymorphism Using Intensional Type Analysis*. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95), San Francisco, California*, S. 130–141, New York, NY. ACM Press.

[Harper und Morrisett 1995b] Harper, Robert und G. Morrisett (1995b). *Compiling polymorphism using intensional type analysis*. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, S. 130–141. ACM Press.

[Hindley 1969] Hindley, R. (1969). *The principal type-scheme of an object in combinatory logic*.

[Hinze 1999a] Hinze, Ralf (1999a). *A Generic Programming Extension for Haskell*. In: Meijer, Erik, Hrsg.: *Proceedings of the 3rd Haskell Workshop, Paris, France*. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-1999-28.

[Hinze 1999b] Hinze, Ralf (1999b). *Polytypic Functions Over Nested Datatypes*. Discrete Mathematics and Theoretical Computer Science, 3(4):193–214.

[Hinze 1999c] Hinze, Ralf (1999c). *Polytypic Functions Over Nested Datatypes (Extended Abstract)*. In: Lins, Rafael Dueire, Hrsg.: *3rd Latin-American Conference on Functional Programming (CLaPF'99)*.

[Hinze 1999d] Hinze, Ralf (1999d). *Polytypic Programming With Ease (Extended Abstract)*. In: Middeldorp, Aart und T. Sato, Hrsg.: *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, Bd. 1722 d. Reihe *Lecture Notes in Computer Science*, S. 21–36. Springer-Verlag.

[Hinze 2000a] Hinze, Ralf (2000a). *Efficient Generalized Folds*. In: Jeuring, Johan, Hrsg.: *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, S. 1–16. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

[Hinze 2000b] Hinze, Ralf (2000b). *Generalizing Generalized Tries*. Journal of Functional Programming, 10(4):327–351.

[Hinze 2000c] Hinze, Ralf (2000c). *Generic Programs and Proofs*. Habilitationsschrift, Universität Bonn.

[Hinze 2000d] Hinze, Ralf (2000d). *Memo functions, polytypically!*. In: Jeuring, Johan, Hrsg.: *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, S. 17–32. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

[Hinze 2000e] Hinze, Ralf (2000e). *A New Approach to Generic Functional Programming*. In: Reps, Thomas W., Hrsg.: *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts, January 19-21*, S. 119–132.

[Hinze 2000f] Hinze, Ralf (2000f). *Polytypic values possess polykinded types*. In: Backhouse, Roland und J. Oliveira, Hrsg.: *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000), July 3-5, 2000*, Bd. 1837 d. Reihe *Lecture Notes in Computer Science*, S. 2–27. Springer-Verlag.

[Hinze 2001a] Hinze, Ralf (2001a). *Manufacturing Datatypes*. Journal of Functional Programming, Special Issue on Algorithmic Aspects of Functional Programming Languages, 11(5):493–524.

[Hinze 2001b] Hinze, Ralf (2001b). *Polytypic Programming With Ease*. Journal of Functional and Logic Programming, 2001(3).

[Hinze 2002] Hinze, Ralf (2002). *Polytypic values possess polykinded types*. Science of Computer Programmming, 43:129–159.

[Hinze 2003] Hinze, Ralf (2003). *Fun with phantom types*. In: Gibbons, Jeremy und O. de Moor, Hrsg.: *The Fun of Programming*, S. 245–262. Palgrave Macmillan. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

[Hinze 2004] Hinze, Ralf (2004). *Generics for the masses*. In: Fisher, Kathleen, Hrsg.: *Proceedings of the 2004 International Conference on Functional Programming, Snowbird, Utah, September 19–22, 2004*. ACM Press.

[Hinze und Jeuring 2003a] Hinze, Ralf und J. Jeuring (2003a). *Generic Haskell: Applications*. In: Backhouse, Roland und J. Gibbons, Hrsg.: *Generic Programming: Advanced Lectures*, Bd. 2793 d. Reihe *Lecture Notes in Computer Science*, S. 57–97. Springer-Verlag.

[Hinze und Jeuring 2003b] Hinze, Ralf und J. Jeuring (2003b). *Generic Haskell: Practice and Theory*. In: Backhouse, Roland und J. Gibbons, Hrsg.: *Generic Programming: Advanced Lectures*, Bd. 2793 d. Reihe *Lecture Notes in Computer Science*, S. 1–56. Springer-Verlag.

[Hinze et al. 2002] Hinze, Ralf, J. Jeuring und A. Löh (2002). *Type-indexed data types*. In: Boiten, Eerke A. und B. Möller, Hrsg.: *Proceedings of the Sixth International Conference on Mathematics of Program Construction (MPC 2002), Dagstuhl, Germany, July 8-10, 2002*, Bd. 2386 d. Reihe *Lecture Notes in Computer Science*, S. 148–174. Springer-Verlag.

[Hinze et al. 2004] Hinze, Ralf, J. Jeuring und A. Löh (2004). *Type-indexed data types*. Science of Computer Programmming, 51:117–151.

[Hinze und Peyton Jones 2001] Hinze, Ralf und S. Peyton Jones (2001). *Derivable Type Classes*. In: Hutton, Graham, Hrsg.: *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, Bd. 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science. The preliminary proceedings appeared as a University of Nottingham technical report.

[Hoogendijk und Backhouse 1997] Hoogendijk, Paul und R. Backhouse (1997). *When do datatypes commute?*. In: Moggi, Eugenio und G. Rosolini, Hrsg.: *Proceedings of the 7th International Conference on Category Theory and Computer Science (Santa Margherita Ligure, Italy, September 4–6)*, Bd. 1290 d. Reihe *Lecture Notes in Computer Science*, S. 242–260. Springer-Verlag.

[Hosoya und Pierce 2003] Hosoya, Haruo und B. C. Pierce (2003). *XDuce: A typed XML processing language*. ACM Transactions on Internet Technology, 3(2):117–148.

[Hughes 1996] Hughes, John (1996). *Type Specialisation for the λ-calculus; or, A New Paradigm for Partial Evaluation Based on Type Inference*. In: Danvy, O., R. Glück und P. Thiemann, Hrsg.: *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, Bd. 1110 d. Reihe *Lecture Notes in Computer Science*, S. 183–215. Springer-Verlag.

[Jansson und Jeuring 1997] Jansson, Patrik und J. Jeuring (1997). *PolyP—a polytypic programming language extension*. In: *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France*, S. 470–482. ACM Press.

[Jansson und Jeuring 2002] Jansson, Patrik und J. Jeuring (2002). *Polytypic Data Conversion Programs*. Science of Computer Programmming, 43(1):35–75.

[Jay et al. 1998] Jay, C.B., G. Bellè und E. Moggi (1998). *Functorial ML*. Journal of Functional Programming, 8(6):573–619.

[Jay und Cocket 1994] Jay, C.B. und J. Cocket (1994). *Shapely types and shape polymorphism*. In: Sanella, D., Hrsg.: *Programming Languages and Systems — ESOP'94: 5th European Symposium on Programming, Edinburgh, UK, Proceedings*, Bd. 788 d. Reihe *Lecture Notes in Computer Science*, S. 302–316, Berlin. Springer-Verlag.

[Jeuring und Jansson 1996] Jeuring, Johan und P. Jansson (1996). *Polytypic Programming*. In: Launchbury, J., E. Meijer und T. Sheard, Hrsg.: *Tutorial Text 2nd International School on Advanced Functional Programming, Olympia, WA, USA*, Bd. 1129 d. Reihe *Lecture Notes in Computer Science*, S. 68–114. Springer-Verlag.

[Jones 1995] Jones, Mark P. (1995). *Functional Programming with Overloading and Higher-Order Polymorphism*. In: Jeuring, J. und E. Meijer, Hrsg.: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, Bd. 925 d. Reihe *Lecture Notes in Computer Science*, S. 97–136. Springer-Verlag.

[Koopman et al. 2003] Koopman, Pieter, A. Alimarine, J. Tretmans und R. Plasmeijer (2003). *GAST: Generic Automated Software Testing*. In: Peña, Ricardo und T. Arts, Hrsg.: *Implementation of Functional Languages: 14th International Workshop, IFL 2002, Madrid, Spain, September 16–18, 2002, Revised Selected Papers*, Bd. 2670 d. Reihe *Lecture Notes in Computer Science*, S. 84–100. Springer-Verlag.

[Lämmel und Peyton Jones 2003] Lämmel, Ralf und S. Peyton Jones (2003). *Scrap your boilerplate: a practical design pattern for generic programming*. ACM SIGPLAN Notices, 38(3):26–37. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[Lämmel und Peyton Jones 2004] Lämmel, Ralf und S. Peyton Jones (2004). *Scrap more boilerplate: reflection, zips, and generalised casts*. Draft; submitted to ICFP 2004.

[Li et al. 2003] Li, Huiqing, C. Reinke und S. Thompson (2003). *Tool support for refactoring functional programs*. In: *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop*, S. 27–38, New York. ACM Press.

[Löh 2004] Löh, Andres (2004). *Exploring Generic Haskell*. Doktorarbeit, Universiteit Utrecht. To appear.

[Löh et al. 2003] Löh, Andres, D. Clarke und J. Jeuring (2003). *Dependency-style Generic Haskell*. In: *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, S. 141–152. ACM Press.

[Meijer et al. 1991] Meijer, E., M. Fokkinga und R. Paterson (1991). *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*. In: *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA*, Bd. 523 d. Reihe *Lecture Notes in Computer Science*, S. 124–144. Springer-Verlag.

[Milner 1978] Milner, Robin (1978). *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences, 17(3):348–375.

[Okasaki 1998] Okasaki, Chris (1998). *Views for Standard ML*. In: *The 1998 ACM SIGPLAN Workshop on ML, Baltimore, Maryland*, S. 14–23.

[Peyton Jones 2003] Peyton Jones, Simon (2003). *Haskell 98 Language and Libraries*. Cambridge University Press.

[Peyton Jones 1996] Peyton Jones, Simon L. (1996). *Compiling Haskell by Program Transformation: A Report from the Trenches*. In: Nielson, Hanne Riis, Hrsg.: *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming, Linköping, Sweden, 22–24 April*, Bd. 1058 d. Reihe *Lecture Notes in Computer Science*, S. 18–44. Springer-Verlag.

[Ruehr 1998] Ruehr, Fritz (1998). *Structural Polymorphism*. In: Backhouse, Roland und T. Sheard, Hrsg.: *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998*. Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ.

[Ruehr 1992] Ruehr, Karl Fritz (1992). *Analytical and Structural Polymorphism Expressed using Patterns over Types*. Doktorarbeit, University of Michigan.

[Sheard 1993] Sheard, Tim (1993). *Type Parametric Programming*. Technischer Bericht CS/E 93-018, Oregon Graduate Institute of Science and Technology, Department of Computer Science and Engineering, Portland, OR, USA.

[Visser 2000] Visser, Eelco (2000). *Language Independent Traversals for Program Transformation*. In: Jeuring, Johan, Hrsg.: *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, S. 86–104. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

[Vytiniotis et al. 2004] Vytiniotis, Dimitrios, G. Washburn und S. Weirich (2004). *An Open and Shut Typecase*. Submitted to ICFP 2004.

[Wadler 1987a] Wadler, P. (1987a). *Views: a way for pattern matching to cohabit with data abstraction*. In: ACM, Hrsg.: *POPL '87. Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of programming languages, January 21–23, 1987, Munich, W. Germany*, S. 307–313, New York, NY, USA. ACM Press.

[Wadler 1988] Wadler, P. (1988). *Deforestation: Transforming Programs to Eliminate Trees*. In: Ganzinger, H., Hrsg.: *Proceedings of the European Symposium on Programming*, Bd. 300 d. Reihe *Lecture Notes in Computer Science*, S. 344–358. Springer Verlag.

[Wadler 1987b] Wadler, Philip (1987b). *Efficient Compilation of Pattern-Matching*, Kap. 5, S. 78–103. Series in Computer Science. Prentice Hall International.

[Wallace und Runciman 1999] Wallace, Malcolm und C. Runciman (1999). *Haskell and XML: Generic combinators or type-based translation?*. In: Lee, Peter, Hrsg.: *Proceedings of the 1999 International Conference on Functional Programm*, S. 148–159.

[Weirich 2001] Weirich, Stephanie (2001). *Encoding Intensional Type Analysis*. In: Sands, David, Hrsg.: *Proceedings of the 10th European Symposium on Programming, ESOP 2001*, Bd. 2028 d. Reihe *Lecture Notes in Computer Science*, S. 92–106.

[Weirich 2002] Weirich, Stephanie (2002). *Higher-Order Intensional Type Analysis*. In: Métayer, Daniel Le, Hrsg.: *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002*, Bd. 2305 d. Reihe *Lecture Notes in Computer Science*, S. 98–114. Springer-Verlag.