Teilprojekt C 1:

NEP:
Programmierumgebung
für nebenläufige Constraints

2.1 Übersicht

Förderungszeitraum:	01. 01. 1999 – 31. 12. 2001
Berichtszeitraum:	01. 01. 1998 – 31. 1. 2001
Projektleiter:	Prof. Dr. Gert Smolka
Dienstanschrift:	FR Informatik Universität des Saarlandes Postfach 15 11 50 66041 Saarbrücken
Telefon:	(0681) 302 – 5311
Telefax:	(0681) 302 – 5341
E-Mail:	smolka@ps.uni-sb.de
Wissenschaftliche Mitarbeiter:	Dipl.-Inform. Leif Kornstaedt Dr. Martin Müller (bis 30.08.1998) Dr. Joachim Niehren (bis 31.12.1998) Dipl.-Inform. Andreas Rossberg (seit 1.06.1998, zeitweise Grundausstattung) Dipl.-Ing. Tobias Müller (Grundausstattung) Dr. Christian Schulte (Grundaustattung)

Ein Hauptziel des Teilprojekts ist die Weiterentwicklung des Programmiersystems Mozart sowie des zugrundeliegenden Paradigmas der nebenläufigen Constraintprogrammierung. Mozart wird in allen C-Projekten des SFBs und im Projekt B1 eingesetzt.

Die wichtigsten Beiträge zur Weiterentwicklung von Mozart im Berichtszeitraum waren: Verbesserung der Constraintkombinatoren und Suchbibliotheken, Fertigstellung des Komponentensystems und der Unterstützung für verteilte Programmierung (Anwendung in B1), Erweiterung der Mengenconstraints (für C2, C3 und C4), sowie first-class constraints (insbesondere für B1). Die diesen Erweiterungen zugrundeliegenden Forschungsarbeiten sind Gegenstand von etwa 20 Veröffentlichungen.

Das zweite Hauptziel des Teilprojekts ist die Entwicklung eines neuen Programmiersystems (Alice), das die Mozart-Funktionalität in verbesserter Form im Rahmen einer statisch getyp-

ten funktionalen Kernsprache zur Verfügung stellt (SML). Die Machbarkeit dieses ambitionierten Vorhabens war zunächst offen, da die Mozart zugrundliegende Sprache relational und dynamisch getypt ist, und Mozart Funktionalität für Constraints, Persistenz und Verteilung zur Verfügung stellt, die es in dieser Form bei funktionalen Sprachen bisher nicht gibt. Die damit verbundenen Fragen konnten positiv beantwortet werden. Als Kernsprache verwenden wir eine Erweiterung von Standard ML. Eine prototypische Implementierung von Alice ist in Arbeit und wird zur Begehung verfügbar sein.

2.2 Kenntnisstand bei der letzten Antragstellung und Ausgangsfragestellung

Der Ausgangspunkt für NEP war das nebenläufige Programmiermodell OPM (Smolka, 1995b, 1995a) und seine praktische Umsetzung durch die Programmiersprache Oz und das Programmiersystem Mozart (Mozart Consortium, 1999a). OPM kombiniert ein nebenläufiges Berechnungsmodell mit hochsprachlichen Primitiven zur Konstruktion von Inferenzkomponenten nach dem Paradigma der Constraintprogrammierung (Van Hentenryck, Saraswat et al., 1997; Marriott & Stuckey, 1998; Wallace, 1996). Constraintprogrammierung kombiniert Ideen aus der logischen Programmierung (Jaffar & Maher, 1994) mit Techniken aus der künstlichen Intelligenz und dem Operation Research und wird erfolgreich für die Lösung kombinatorischer Optimierungsprobleme eingesetzt. Constrainttechniken eignen sich insbesondere auch für die syntaktische und semantische Verarbeitung natürlicher Sprache, was sie für den SFB, insbesondere für die Teilprojekte des C-Bereichs, interessant macht.

Oz wurde von der Gruppe des Antragsstellers zunächst am DFKI (1991-1998) entwickelt. Seit 1996 trägt der SFB maßgeblich zur Weiterentwicklung von Oz und seiner praktischen Realisierung im Programmiersystem Mozart bei. Im Rahmen des Mozart Konsortiums (Mozart Consortium, 1999b) sind zwei weitere Forschungsgruppen (Louvain und SICS) an der Entwicklung von Mozart beteiligt.

Mozart ist nicht einseitig auf nebenläufige Constraintprogrammierung spezialisiert. Eine zweite Stoßrichtung betrifft die hochsprachliche Unterstützung von Persistenz und verteilter Programmierung im Kontext des Internets. Hier spielen Techniken für das Programmieren im Großen eine wichtige Rolle. Diese Seite von Mozart spielt bei der verteilten Mathematik-Umgebung, die im Teilprojekt OMEGA (B1) entwickelt wird, eine wichtige Rolle.

Ein wichtiger Teil der Ausgangsfragestellung betrifft die Weiterentwicklung von Mozart. Zum einen sollten das Komponentensystem und die Verteilungsschicht fertiggestellt werden (mit

Anwendungen in B1). Zum anderen sollte die Constrainttechnologie verbessert und erweitert werden, insbesondere die Bereiche Suche, Constraintkombinatoren und Mengenconstraints (mit Anwendungen in C2, C3 und C4).

Der zweite Teil der Ausgangsfragestellung betrifft die theoretische Weiterentwicklung von Constraintsystemen, insbesondere für statische Programmanalyse und Fehlerdiagnose. Hier geht es vor allem um Mengen- und Subsumptionsconstraints.

Der dritte Teil der Ausgangsfragestellung ist längerfristiger angelegt. Hier geht es um die Rekonstruktion der Mozart-Funktionalität im Rahmen einer statisch getypten funktionalen Kernsprache. Statische Typisierung bedeutet, dass die Programmiersprache eine Spezifikationssprache für Schnittstellen beinhaltet. Die Einhaltung der spezifizierten Schnittstellen wird bei der Programmerstellung automatisch verifiziert (also vor der Programmausführung). Durch statische Typisierung erreicht man eine verbesserte Programmqualität und eine signifikante Reduzierung des Aufwands für die Entwicklung, Wartung und Erweiterung von Software. Statische Typisierung ist ein Leitprinzip, das einen grundlegenden Einfluß auf den Design einer Programmiersprache hat. Daher ist ein nachträgliches Hinzufügen von statischer Typisierung zu einem bestehendem Design (wie Oz), das ohne dieses Leitprinzip entwickelt wurde, unmöglich.

2.3 Angewandte Methoden

Die Aufgabenstellung des Projektes erfordert ein sehr breites Methodenspektrum. Für die Realisierung von Mozart benötigt man Ingenieurs-Methoden aus den Bereichen Programmiersysteme und Betriebssysteme. Bei der Weiterentwicklung und Anwendung der Constraint-Technolgie kommen Methoden aus der Algorithmik, der Künstlichen Intelligenz und dem Operations Research zum Einsatz. Für die theoretische Untersuchung von Constraintsysteme benötigt man Methoden aus der computationalen Logik und der Komplexitätsanalyse. Für die Untersuchung der programmiersprachlichlichen Aspekte benötigt man die Theorie der Programmiersprachen.

2.4 Ergebnisse und ihre Bedeutung

Ergebnisse wurden erzielt bei der Weiterentwicklung von Mozart, bei der theoretischen Untersuchung von Constraintsystemen und Berechnungsmodellen, beim Design und der Implementierung von Alice, sowie bei der Kooperation mit anderen Teilprojekten des SFBs.

2.4.1 Weiterentwicklung von Mozart

Suche und Constraintkombinatoren

Suche. Constraint-Propagierung alleine reicht nicht aus, um ein Constraintproblem zu lösen. Um den Lösungsprozess weiter voranzutreiben, wird ein Problem durch Fallunterscheidung in Teilprobleme zerlegt. Die Zerlegung reichert die Teilprobleme um Constraints an, die eine weitere Constraintpropagierung anstoßen. Suche iteriert Propagierung und Zerlegung und spannt dabei einen Suchbaum auf.

Fallunterscheidungen definieren den Suchbaum. Orthogonal dazu ist, wie der resultierende Suchbaum durch eine Suchmaschine exploriert wird. Traditional ist Zerlegung programmierbar, Exploration nicht. Oz überwindet diese Restriktion und erlaubt die Programmierung von Suchmaschinen. Exploration wird programmierbar, indem die Knoten eines Suchbaumes explizit durch emanzipierte Berechnungsräume (First-class Computation Spaces) in Oz repräsentiert sind (Schulte, 1997a, 2000b). Berechnungsräume werden als ein abstrakter Datentyp realisiert.

Im Berichtszeitraum haben sich die Arbeiten darauf konzentriert, die Grundidee von programmierbarer Exploration in Richtung Effizienz und breiterer Anwendbarkeit voranzutreiben.

Effizienz. Suche basiert in Oz auf Kopieren, da der traditionelle Ansatz des Zurücksetzens unverträglich mit Nebenläufigkeit ist. Adapative Wiederberechnung wurde als hybride Technik entwickelt, die Kopieren mit Wiederberechnung kombiniert (Schulte, 1999, 2000b). Adaptive Wiederberechnung ist automatisch; sie bedarf keiner Parametrisierung durch den Benutzer. Es wurde gezeigt, daß Oz mit adaptiver Wiederberechnung bei großen Constraint-Problemen kommerziellen Constraintprogrammiersystemen, die auf Zurücksetzen basieren, bezüglich Laufzeit und Speicher überlegen ist.

Ein zweite Stoßrichtung, um Suche effizienter zu machen, ist die parallele Exploration eines Suchbaumes auf mehreren vernetzten Rechnern. Es wurde eine verteilte Suchmaschine entwickelt, die durch Verteilung und programmierbare Suche in Oz einfach zu realisieren war (Schulte, 2000a, 2000b). Der Aufwand dafür liegt bei weniger als 1000 Zeilen Oz. Die Maschine erreicht einen befriedigenden, oftmals linearen Speedup. Mozart ist zur Zeit das einzige Constraintprogrammiersystem, das parallele Suche unterstützt.

Constraintkombinatoren. Neben Suche wurden Berechnungsräume auch auf das Programmieren von kompositionalen Kombinatoren angewendet (Schulte, 2000b). Kompositionale

Constraintkombinatoren (Deep-Guard Combinators) haben sich als unverzichtbar für die Teilprojekte C3 und C4 erwiesen. Die auf Berechnungsräumen basierenden Kombinatoren haben bei gleicher Effizienz eine dramatisch einfachere Implementierung als die bisherige Realisierung mit C++ (Janson, 1994; Mehl, 1999). Für ein besseres Verständnis der Typisierung wurden Berechnungsräume auch im Kontext von Alice modelliert (Schulte, 2000c).

Erweiterung der Constraint-Funktionalität

First-Class-Constraints. Das Oz Programming Model (OPM, Smolka, 1995b) unterscheidet zwischen *Basis-Constraints* und *höheren Constraints*. Basis-Constraints definieren den Wertebereich einer Variablen. Höhere Constraints werden durch *Propagierer* implementiert, die durch Constraint-Propagierung den Wertebereich ihrer Variablen einschränken.

Oft ist eine wesentlich wirkungsvollere Einschränkung – und damit eine Einschränkung des Suchraumes – möglich, wenn man Wissen über höhere Constraints direkt einfließen lassen kann. Nehmen wir folgende Constraints an: $x_1 \neq x_2 \wedge x_1{+}1 = x_3 \wedge x_2{+}1 = x_3$. Im Gegensatz zu Inferenz über höheren Constraints, ist traditionelle Constraintpropagierung nicht in der Lage, Unerfüllbarkeit dieser Constraints zu finden. Diese Beobachtung führte zur Einführung von Constraints als emanzipierte Datenstrukturen (first-class constraints), die neue Propagierungstechniken ermöglicht (Müller, 2000b). Die beschriebenen Techniken wurden im Projekt OMEGA auf dem Gebiet des automatischen Beweisplanens erfolgreich angewendet (Melis, Zimmer & Müller, 2000a, 2000b).

Constraint Propagator Interface. Die Flexibilität von Constraintlösern wird wesentlich erhöht, wenn die Benutzer des Lösers ihre eigenen Propagierer implementieren und benutzen können. Um dieses zu ermöglichen, wurde eine Schnittstelle, das *Constraint Propagator Interface*, kurz CPI, entworfen und implementiert (Müller & Würtz, 1999).

Der Kern eines Propagierers ist sein *Filter*, der die Wertebereiche der Variablen einschränkt. Im Gegensatz zu Propagierern sind Filter unabhängig von der konkreten Implementierung eines Constraintprogrammiersystems. Daher wurde zusätzlich eine Schnittstelle für Filter entwickelt (Ng, Choi, Henz & Müller, 2000), die deren Austausch zwischen verschiedenen Constraintprogrammiersystemen ermöglicht.

Mengenconstraints. Die bereits im vorherigen Berichtszeitraum begonnene Implementierung von Mengenconstraints (endliche Mengen von natürlichen Zahlen) in Oz wurde vervollständigt. Insbesondere wurde eine Generalisierung des Element-Constraints (Van Henten-

ryck, Simonis & Dincbas, 1992) für Mengen realisiert. Es hat sich in vielen Anwendungen in den Teilprojekten C3 und C4 gezeigt, dass sich Mengenconstraints im allgemeinen und der Element-Constraint über Mengen im speziellen als eine ausgezeichnete Grundlage für elegante und effiziente Formalisierungen eignen.

Verteilung

Mit der steigenden Bedeutung des Internets und der zunehmenden Verbreitung von Rechnernetzwerken rückt auch die Entwicklung verteilter Anwendungen immer mehr in den Mittelpunkt. Verteilung ist zudem ein zentrales Konzept, um ressourcenadaptive Prozesse zu verwirklichen: durch Zerlegung eines Problems in kleine Teile, die dynamisch auf verschiedene Rechner verteilt werden, können vorhandene Rechenkapazitäten optimal ausgenutzt werden. Allerdings gilt verteilte Programmierung nach wie vor als schwierig.

Oz war von Anbeginn an als nebenläufige Programmiersprache konzipiert. Die Sprache stellte auf Grund dieser und anderer Eigenschaften einen idealen Ausgangspunkt dar, um verteilte Programmiermodelle zu untersuchen. Im Berichtszeitraum wurden in Zusammenarbeit mit dem Swedish Institute of Computer Science die im PERDIO-Projekt entwickelten Ansätze für ein verteiltes Oz zu Ende geführt.

Es wurden die zentralen Aspekte verteilter Programmierung – Verteilungsstruktur, Fehlertoleranz, Sicherheit und Offenheit – isoliert und ein Modell für verteiltes Oz entwickelt, welches diese Aspekte behandelt, indem es das zentralisierte OPM um geeignete Sprachmittel ergänzt (Haridi, Van Roy, Brand & Schulte, 1998). Das Modell erlaubt verteilte Programmierung auf einer hohen Abstraktionsebene. Es erlaubt die vollständige Entkopplung der Programmfunktionalität von Aspekten der Verteilung.

Die in Abschnitt 2.4.1 diskutierten parallelen Suchmaschinen sind eine überzeugende Fallstudie für verteilte Programmierung in Oz. Die von Schulte (2000a) entwickelte Architektur basiert neben programmierbarer Suche auch entscheidend auf der Möglichkeit, Funktionalität orthogonal zu Verteilung zu betrachten.

Synchronisation nebenläufiger Prozesse erfolgt in Oz mit Hilfe logischer Variablen. Ein verteiltes Programmiermodell erfordert deshalb eine transparente verteilte Semantik logischer Variablen und damit verteilte Unifikation. Ein entsprechender Algorithmus wurde entwickelt und seine Korrektheit gezeigt (Haridi et al., 1999). Darüber hinaus wurde demonstriert, wie bekannte Idiome verteilter Programmierung mit Hilfe verteilter logischer Variablen ausge-

drückt werden können. Sie erlauben dabei auf elegante und für den Programmierer leicht handhabbare Weise ein hohes Maß an Toleranz gegenüber Netzwerklatenz.

Komponenten

Mozart wurde um ein dynamisches Modulsystem erweitert (Duchier, Kornstaedt, Schulte & Smolka, 1998). Das System erlaubt getrennte Übersetzung sowie dynamisches und statisches Binden von Komponenten. Komponenten werden in Mozart als Funktoren bezeichnet. Funktoren können von beliebigen Internet-Adressen geladen werden und sind über Plattformgrenzen hinweg portabel. Funktoren können dynamisch zur Laufzeit erzeugt werden und können damit beliebige vorberechnete Oz-Datenstrukturen enthalten.

Neben der Strukturierung von Software bildet das Modell auch die Grundlage für die Erzeugung und Kontrolle von verteilten Compute-Servern. Entscheidend ist dabei, dass durch Binden von Komponenten ein kontrollierter Zugriff auf die Ressourcen eines anderen Rechners erreicht wird. Direkt damit zusammenhängend erlaubt das Modell auch die Umsetzung verschiedener Sicherheitsstrategien, um Funktoren auch von nicht vertrauenswürdigen Quellen sicher ausführen zu können.

Das Mozart-System implementiert dieses Komponentenmodell in vollem Umfang. In Oz geschriebene Softwaresysteme lassen sich dadurch in hohem Maße dynamisch konfigurieren. Das MOGUL-Archiv (Duchier, 2000) ermöglicht es Oz-Benutzern, Komponenten zur Verfügung zu stellen, und bietet eine Infrastruktur zur Verwaltung solcher Komponenten. Dies stellt eine wertvolle Ressource dar, die innerhalb des SFB insbesondere von den Projekten OMEGA (B1) und CHORUS (C4) verwendet wird.

Werkzeuge

Visualisierer für Constraint-Graphen. Neben der Effizienz ist die Korrektheit selbst eines der größten Probleme beim Lösen von Constraintproblemen. Implementiert der Constraintlöser das Problem, d.h. machen die gefunden Lösungen Sinn? Falls das nicht der Fall ist, welche Constraints sind nicht korrekt? Die Analyse fehlerhafter Constraintlöser wurde bisher nicht unterstützt. In (Müller, 2000a) wird ein graphbasierter Ansatz vorgeschlagen und demonstriert. Das daraus abgeleitete Werkzeug, der *Constraint Investigator*, wurde prototypisch implementiert. Die Idee des Ansatzes ist es, den Zustand eines Constraintlösers durch verschiedene Graphen darzustellen. Zwischen den Graphdarstellungen kann der Benutzer belie-

big wechseln. Der Investigator ergänzt damit den Explorer (Schulte, 1997b) um ein wichtige Komponente zum Debugging von Constraintproblemen.

Inspector. Der *Inspector* (Brunklaus, 2000) ist ein im Rahmen einer Diplomarbeit entwickeltes Tool zur graphischen Darstellung von Oz-Datenstrukturen. Er erlaubt die effiziente und flexible, interaktive Anzeige beliebig komplexer Oz-Datenstrukturen.

Oz Profiler. Zur Untersuchung und Optimierung des Ressourcenverbrauchs von Oz-Programmen wurde ein Profiler entwickelt, welcher den Speicherverbrauch und das Laufzeitverhalten von Programmen darstellen kann.

2.4.2 Theoretische Arbeiten

Mengenconstraints. Constraints, die Relationen zwischen Mengen von Bäumen beschreiben, werden seit Anfang der 90iger Jahre intensiv im Bereich der Programmanalyse untersucht (Aiken, Kozen & Wimmers, 1995; Gilleron, Tison & Tommasi, 1993; Aiken, Kozen, Vardi & Wimmers, 1993; Charatonik & Podelski, 1996). Atomare Mengenconstraints sind einfache Inklusionen $t_1 \subseteq t_2$ zwischen üblichen Termen ohne Mengenoperatoren. Hier konnten (Niehren, Müller & Talbot, 1999) einen Fehler in einer neueren wissenschaftlichen Publikation finden und korrigieren. Sie zeigten, dass das Entailment-Problem für atomare Mengenconstraints nicht etwa polynomial, sondern PSPACE-vollständig ist.

Subsumptionsconstraints. Featurebäume sind unter verschiedenen Bezeichnungen allerseits in mehreren Gebieten bekannnt: Sie heißen Modul- oder Objekttypen in der Typisierung, Records oder Module in der Programmierung, und Syntaxbäume in der Computerlinguistik. Die natürliche Ordnung auf Featurebäumen hat entsprechend viele Namen. Je nach Bereich heißt sie Teiltypisierung, Informationsordnung, oder Subsumption.

Subsumption wird in der computerlinguistischen Modellierung mit HPSG seit langem verwendet (Pollard & Sag, 1987), wurden aber erst von Dörre (1991) explizit über Featurebäumen interpretiert. Müller, Niehren und Podelski (2000) konnten nun erstmals einen kubischen Algorithmus angeben, der die Erfüllbarkeit von Subsumptionconstraints testet und zusätzlich sogar Entailment entscheiden kann.

Die Situation wird drastisch komplexer, wenn man beim Entailment von Subsumptionsconstraints Existenzquantoren hinzunimmt, also $\varphi \models^? \exists X_1 \ldots \exists x_n \varphi'$ betrachtet. Für diese schwere Problem konnten Müller und Niehren (1998), Müller und Niehren (2000) erstmal die Ent-

scheidbarkeit nachweisen, indem sie eine neue Beziehung zur zweitstufigen monadischen Logik über Wörtern (S2S) ausnutzen. Müller, Niehren und Treinen (1998) konnten darüberhinaus die genaue Komplexitätklasse (PSPACE-vollständig) bestimmen. Sie konnten auch zeigen, dass die Theorie der ersten Stufe der Subsumption (im Gegensatz zur Gleichheit) über Featurebäumen unentscheidbar ist.

Eine weitere schwierige Entailmentvariante ist als "nicht-strukturelles Teiltypentailment" bekannt (Eifrig, Smith & Trifonow, 1995; Pottier, 1996; Henglein & Rehof, 1997). Es tritt bei erweiterten Subsumptionsconstraints auf, mit denen man die Stelligkeit eines Baumkonstruktors in der Constraintsprache beschränken kann. Die Entscheidbarkeit von nicht-strukturelles Teiltypentailment ist seit langem offen. Niehren und Priesnitz (1999) zeigten nun, dass ein interessantes Teilproblem (ohne Existenzquantoren) bereits PSPACE-vollständig ist. Trotz erheblicher Fortschritte (Niehren & Priesnitz, 2001) konnte sie das Gesamtproblem bislang aber nicht lösen.

Um objektorientierte Ausdrucksmittel flexibel zu behandeln, wurde Typinferenz für emanzipierte Methoden (first-class messages) untersucht. Ein Constraintsystem auf der Basis von Subsumptions-Constraints wurde entwickelt, welches polymorphe Typinferenz für solche Konstrukte ermöglicht. Monomorphe Typinferenz in diesem System ist in polynomialer Zeit möglich (Müller & Nishimura, 1998, 2000).

Fehlerdiagnose. Für eine Teilsprache von Oz wurde ein Diagnosesystem entwickelt, welches durch Programmanalyse versucht, eine möglichst große Klasse von Programmierfehlern automatisch zu identifizieren. Da Oz reich an Konstrukten ist, die nicht nach einer strengen Typdisziplin behandelt werden können, ist eine solche Analyse notwendigerweise konservativ approximativ. Es werden dazu nach einem kompositionalen Schema zu einem Programm Constraints generiert, die den möglichen Datenfluss beschreiben. Sind diese Constraints unerfüllbar, so ist das Programm als fehlerhaft diagnostiziert (Müller, 1998; Podelski, Charatonik & Müller, 1999).

Nebenläufige Programmiermodelle. Die formale Untersuchung des Oz-Programmiermodels (OPM) von Smolka (1995b), wie z.B. im γ-Kalkül (Smolka, 1994) konzentrierte sich bereits in der ersten Phase von NEP auf das Kernmodel für die nebenläufige Programmierung höherer Stufe und betrachtete Constraintprogrammierung stets als orthogonale Ergänzung.

Das OPM ist im Kern relational aufgebaut, wie bereits oben diskutiert. In einem relationen Ansatz stellte sich natürlich die Frage, ob und wie sich funktionale Elemente ausdrücken und auszeichnen lassen. Unabhängig davon stellt sich auch die verwandte Frage nach der

genauen Beziehung des OPM zu anderen Modellen der nebenläufiger Berechnung, wie zum Beispiel dem weithin anerkannten π-Kalkül (Milner, Parrow & Walker, 1992). Denn sowohl die strikte, als auch die nicht-strikte funktionale Berechnung lassen sich bekannterweise im π-Kalkül ausdrücken (Milner, 1992; Brock & Ostheimer, 1995). Das dazu verwendete Fragment des π-Kalkül ist allerdings schwer charakterisierbar, weder syntaktisch noch semantisch.

Im Berichtszeitraum zeichnete Niehren (2000) nun den *uniform konfluenten* Kern des OPM aus und zeigte, dass und wie sich die funktionale Programmierung (strikt und nicht-strikt) darin modellieren läßt. Er konnte auch nachweisen, dass dieser Kern des OPM, der sogar leicht eingeschränkte logische Variablen umfaßt, in den π-Kalkül eingebettet werden kann. Basierend auf dem von ihm eingeführten Begriff der uniformen Konfluenz konnte Niehren letztendlich eine lange bekannte Beziehung zwischen strikter und nicht-strikter funktionaler Programmierung im Rahmen der nebenläufigen Berechnung erstmals formal nachweisen: Nicht-strikte funktionale Berechnung ist stets mindestens so zeiteffizient wie die strikte, unter der Bedingung, dass man den zusätzliche Verwaltungsaufwand ignoriert.

2.4.3 Entwicklung von Alice

Mozart ist das Ergebnis eines ambitionierten Forschungsprojektes, das 1991 am DFKI gestartet wurde. Das Hauptziel dabei war, die zu diesem Zeitpunkt nur in Ansätzen vorhandene nebenläufige Constrainttechnologie praktisch weiter zu entwickeln und im Rahmen eines Programmiersystems für nichttriviale Anwendungen bereitzustellen. Das Paradigma der nebenläufigen Constraintprogrammierung erwies sich in mehreren Teilprojekten des SFBs als wichtige Leitidee und hatte nachhaltig Einfluß auf deren Ausrichtung. Eine unabdingbare Voraussetzung dafür war die softwaretechnische Realisierung des Paradigmas durch das Programmiersystem Mozart.

Ab 1995 ging unsere Arbeitsgruppe mit dem PERDIO-Projekt eine zweite, eigentlich unabhängige Aufgabe an: die Unterstützung von verteilter Programmierung im Internet. Dafür konnten wir Partner in Belgien (Universität Louvain) und Schweden (SICS) gewinnen, mit denen wir seitdem im Mozart-Konsortium zusammenarbeiten. Heute ist Mozart ein voll ausgebautes Programmiersystem, das sowohl nebenläufige Constraints als auch verteilte Programmierung im Internet unterstützt. Mit Mozart können Anwender heute anspruchsvolle verteilte Systeme zu bauen, ohne dafür die komplexen Details der Betriebssystemsebene verstehen zu müssen. Davon hat im SFB inbesondere das Teilprojekt OMEGA profitiert.

Mozart ist also das Ergebnis von explorativer Forschung, bei der eine große Bandbreite von

Zielen verfolgt wurde. Trotzdem wurde ein voll anwendungstaugliches Programmiersystem entwickelt, dass seit 1995 weltweit benutzt wird. Mittlerweile wurden viele Anwendungen mit Mozart realisiert, auch außerhalb des akademischen Bereichs.

Die Anwendungen und die Benutzergemeinde haben gezeigt, dass die duch Mozart realisierte Funktionalität höchst attraktiv und nachgefragt ist. Mozart ist die erste Realisierung dieser Funktionalität, aber nicht die bestmögliche. Es gibt viele Möglichkeiten für signifikante Vereinfachungen und Verbesserungen. Diese erfordern jedoch radikale Designänderungen, die nicht konservativ zu Mozart hinzugefügt werden können. Der effektivste Weg, den mit Mozart erreichten Stand weiter zu entwickeln, ist das Design und die Implementierung eines neuen Programmiersystems. Damit haben wir gemäß der Antragsstellung begonnen. Das zu entwickelnde System haben wir Alice getauft.

Sprachentwurf

Alice (als Programmiersprache) haben wir als konservative Erweiterung der statisch getypten funktionalen Sprache Standard ML (SML) (Milner, Tofte, Harper & MacQueen, 1997) konzipiert. Damit können wir auf die Ergebnisse jahrzentelanger Forschung im Bereich funktionale Programmierung und Typsysteme zurückgreifen. Im Vergleich zu Mozart hat das drei gravierende Konsequenzen:

1. **Funktionale Kernsprache.** Das ist ein radikaler Bruch mit Mozart, das wie logische Programmiersprachen (der Ausgangspunkt für Mozart), auf einer relationalen Kernsprache mit logischen Variablen basiert. Dieser Bruch entspricht unser Anwendungserfahrung mit Mozart, die klar zeigt, dass der dominierende Programmiermodus funktional ist. Logische Variablen sind eine wertvolle Ergänzung, sollten aber nicht für Zwecke missbraucht werden, die sich einfacher funktional formulieren lassen. Die Argumente für eine funktionale Kernsprache werden in (Smolka, 1998) formuliert.
2. **Statische Typisierung.** Statische Typisierung bedeutet, dass die Programmiersprache eine Spezifikationssprache für Schnittstellen beinhaltet. Die Einhaltung der spezifizierten Schnittstellen wird bei der Programmerstellung automatisch verifiziert (also vor der Programmausführung). Durch statische Typisierung erreicht man eine verbesserte Programmqualität und eine signifikante Reduzierung des Aufwands für die Entwicklung, Wartung und Erweiterung von Software. Statische Typisierung ist ein Leitprinzip, das einen grundlegenden Einfluß auf den Design einer Programmiersprache hat. Daher ist ein nachträgliches Hinzufügen von statischer Typisierung zu einem bestehendem Design, der

ohne dieses Leitprinzip entwickelt wurde, unmöglich.

3. **Konservative Erweiterung einer etablierten Programmiersprache.** Da Alice eine konservative Erweiterung von SML ist, müssen Anwender keine neue Programmiersprache (Syntax, Typsystem) lernen. Außerdem kann man man SML-Programme sofort als Alice-Programme benutzen. Das ist ein großer Vorteil gegenüber Oz, dass seine eigene idiosynkratische Syntax hat.

Ein signifikanter Teil der Mozart-Funktionalität kann jetzt konservativ und ohne neue Syntax hinzugenommen werden:

1. **Futures und Threads.** Futures sind Platzhalter für Werte und stellen eine Verfeinerung von logischen Variablen dar. Zusammen mit Threads werden damit die nebenläufigen Programmiertechniken aus Mozart verfügbar. Mit By-need-Futures kann bedarfsgesteuerte Auswertung wie bei nichtstrikten funktionalen Sprachen (zum Beispiel Haskell) modelliert werden. Das Konzept der “gescheiterten Future” ermöglicht eine saubere Kombination von Ausnahmen und bedarfsgesteuerter Auswertung (eine Verbesserung gegenüber Mozart). Dieser Teil von Alice wird in (Smolka, 1999) erklärt.
2. **Berechnungsräume.** Berechnungsräume sind die Abstraktion, mit der in Mozart Suche und Constraintkombinatoren programmiert werden. Diese Abstraktion ist im wesentlichen sprachunabhängig und lässt sich problemlos mit der Typstruktur von SML modellieren (Schulte, 2000c).
3. **Finite Domain und Finite Set Constraints.** Diese Constraintsysteme können relativ einfach als SML-Strukturen verfügbar gemacht werden. Die Constraintvariablen werden als Werte von abstrakten Typen modelliert. Der Übergang zu Futures ist explizit.

Schwieriger war die Integration der Persistenz- und Verteilungskonzepte von Mozart. Diese Konzepte benötigen Schnittstellen mit dynamischer Typprüfung, die es in SML nicht gibt. Eine dynamische Typprüfung ist immer notwendig, wenn ein Prozess eine externe Struktur importiert. Die externe Struktur kann aus einem persistenten Speicher (ein Dateisystem) oder aus einem anderem Prozess (Verteilung) importiert werden. Hier haben wir eine elegante Lösung gefunden, die die Typaspekte und die Importoperationen orthogonalisiert. Dazu führen wir zwei neue Operationen pack und unpack ein. Pack konvertiert Module in so genannte Pakete. Dabei handelt es sich um die Werte des speziellen Typs package. Module können alle sprachlichen Objekte umfassen. Unpack erledigt die Rückführung von Paketen in Module. Dabei muss die Schnittstelle spezifiziert werden, mit der das Modul sichtbar sein soll. Um die Zulässigkeit der Schnittstelle sicherzustellen, ist eine dynamische Typprüfung erforderlich.

Unpack ist das einzige Sprachprimitiv, für das eine dynamische Typprüfung erforderlich ist. Import und Export von sprachlichen Objekten erfolgt jetzt auf der Ebene von Paketen, also völlig orthogonal zur Typisierung. Damit diese Konstruktion hinreichend flexibel ist, benötigt Alice ein höherstufiges Modulsystem. Höherstufige Modulsysteme sind theoretisch hinreichend verstanden und sind im Rahmen des ML-Dialekts OCaml (INRIA) praktisch erprobt.

In Bezug auf Verteilung soll Alice deutlich einfacher als Mozart werden, das hier ein sehr reiches Repertoire zur Verfügung stellen. Im Wesentlichen soll eine allgemeine Form des Remote Procedure Calls zur Verfügung gestellt werden, bei der fast alle sprachlichen Objekte als Argumente und Ergebnisse übergeben werden können. Die Erfahrung mit Mozart hat gezeigt, dass damit die meisten Anwendungen bequem realisiert werden können. Das gegenüber Mozart stark vereinfachte Modell hat den Vorteil, dass es sich einfacher implementieren und warten lässt. Außerdem lassen sich mit diesem Modell verteilte Anwendungen, die den Ausfall einzelner Teilprozesse tolerieren, einfacher realisieren.

Die Modellierung von Baumconstraints ist nocht nicht geklärt.

Oz unterstützt objektorientierte Programmierung mithilfe eines flexiblen Objektsystems. Dieses wird auch von vielen Anwendungen benutzt. Nach einigen Versuchen kamen wir zu dem Schluß, dass SML durch die Integration eines Objektsystems mehr verliert als gewinnt, da das Objektsystem viele Konzepte der funktionalen Kernsprache duplizieren muss und die Komplexität der Sprache gewaltig erhöht (siehe OCAML). Für Anwender ist es zudem unserer Einschätzung nach fast immer von Vorteil, wenn die Sprache einen klaren Programmierstil vorgibt, statt die Qual der Wahl zu lassen.

Implementierung

Alice wird wie Mozart mit Hilfe eines Übersetzers und einer virtuellen Maschine (VM) realisiert. Die VM kennt nur Objekte der Art Wert. Objekte anderer Arten (zum Beispiel Typen und Module) werden durch den Übersetzer auf Werte zurückgeführt. Die durch die VM realisierte Sprache ist also sehr viel einfacher als Alice. Zur Zeit arbeiten wir mit einer geringfügig erweiterten Version der Mozart VM. Damit können wir in vollem Umfang auf die für Mozart implementierte Funktionalität zurückgreifen (zum Beispiel logische Variablen, Threads, Constraints, Spaces, Pickling, Inter-Prozess-Kommunikation). Außerdem können wir in Oz geschriebene Software Komponenten mit in Alice geschriebenen Komponenten kombinieren.

Eine erste, prototypische Alice-Implementierung ist bereits lauffähig. Diese beinhaltet ge-

trennte Übersetzung, bedarfsgesteuertes Laden von Komponenten und einen Interpreter. Bis auf Inter-Prozess-Kommunikation sind die oben aufgeführten Konzepte realisiert. Die Implementierung soll als Grundlage für Pilotanwendungen dienen. Diese betreffen inbesondere die Constraint-Bibliotheken, die Grafikbibliothek und verschiedene Programmierwerkzeuge.

Die Homepage für Alice hat die Adresse www.ps.uni-sb.de/alice/.

2.4.4 Kooperationen innerhalb des Sonderforschungsbereichs

Kooperation mit dem Teilprojekt B1 (OMEGA). Ein Ziel des Teilprojektes B1 (OMEGA) ist die Integration von Constraintlösern in die Beweisplanung. Hier entstand eine fruchtbare Zusammenarbeit, die zu einer gemeinsam betreuten Diplomarbeit (Zimmer, 2000) und gemeinsamen Publikationen führte (Melis et al., 2000a, 2000b). Auf der Grundlage von Mozart wurde der Constraintlöser CoSIE entwickelt, der jetzt Bestandteil des OMEGA-Systems ist. Mozart wurde für diesen Zweck um einen Löser für Interval-Constraints und sogenannte First-Class-Constraints erweitert.

Der entwickelte Constraintlöser CoSIE sammelt die während des Beweisplanens anfallende Constraints auf und repräsentiert sie als abfragbare Datenstrukturen. Dadurch sind gleichzeitig numerische und symbolische Inferenzen auf den Constraints möglich und der Beweisplaner kann erfolglose Beweispläne frühzeitig verwerfen. Nachdem alle Constraints vorhanden sind, sucht der Constraintlöser symbolische Instanzen für die auftretenden Problemvariablen, die die Constraints erfüllen. Diese werden dem Beweisplaner übergeben, der sie in den Beweisplan integriert.

Ein zweiter enger Kooperationsbereich ist die von OMEGA auf der Basis von Mozart entwickelte verteilte Mathematikumgebung, die die Persistenz- und Internet-Funktionalität von Mozart voll ausreizt. Hier war Mozart zunächst einmal die technische Basis, auf der dieses ambitionierte System mit vergleichsweise geringen Ressourcen und begrenzter System-Expertise realisiert werden konnte. Mozart macht fortgeschrittene Internet-Funktionalität auf einer hohen Ebene verfügbar, die die komplexen Details der tieferen Systemebenen verbirgt. Von unserer Seite wurde vor allem softwaretechnische Unterstützung für den Einsatz der durch Mozart unterstützten Internet-Technologien beigesteuert.

Kooperation mit dem Teilprojekt C3 (NEGRA). Ein Hauptziel von NEGRA ist die Entwicklung von effizienten, constraint-basierten Parsingtechniken für Dependenz-Grammatiken. Dabei wird die Constraint-Funktionalität im vollen Umfang zum Einsatz gebracht. Insbeson-

dere sind hier die im Berichtszeitraum verbesserten Suchprimitive und Constraintkombinatoren von Bedeutung. NEGRA war das erste Projekt, das die zunächst nur im beschränkten Umfang in Mozart realisierten Mengenconstraints als zentrale Technologie zum Einsatz brachte. Es stellte sich heraus, dass Mengenconstraints für viele Aufgabenstellungen als effiziente Inferenztechnologie eingesetzt werden können. Mittlerweile werden Mengenconstraints auch in LISA (C2) und CHORUS (C4) an maßgeblicher Stelle eingesetzt. NEP sorgte dafür, dass Mengenconstraints in Mozart effizient und mit der für die Anwendungen notwendigen Abdeckung und Funktionalität implementiert wurden.

Für die Realisierung der Mengenconstraints und des darauf aufbauenden Selektionsconstraints war die Weiterentwicklung der CPI-Schnittstelle von Mozart von entscheidender Bedeutung. Mit dieser Schnittstelle können Benutzer neue Constraints zu Mozart hinzufügen, ohne mit den komplexen Interna von Mozart vertraut zu sein. In diesem Zusammenhang wird auch die Technologie für portable native Funktoren benötigt, die im Berichtszeitraum von NEP entwickelt wurde.

Constraints werden durch Propagierungsalgorithmen realisiert, die in C++ programmiert und über die CPI-Schnittstelle in Mozart eingebracht werden. Die Propagierungsalgorithmen lassen sich insbesondere bei Mengenconstraints durch Propagierungsregeln beschreiben. NEP und NEGRA entwickeln gemeinsam einen Übersetzer, der einen durch Regeln spezifizierten Propagierungsalgorithmus automatisch nach C++ übersetzt. Alle Propagierungsalgorithmen für Mengenconstraints sollen mithilfe dieses Übersetzers implementiert werden. Dadurch kann einerseits zeitaufwendige Programmierung eingespart werden. Andererseits ist es auf der Ebene der Regeln sehr viel einfacher als auf der Ebene von C++, die Korrektheit der Propagierungsprogramme sicherzustellen.

Kooperation mit dem Teilprojekt C4 (CHORUS). Ein Leitgedanke von CHORUS ist die Modellierung von semantischer Unterspezifikation mit Hilfe von Constraints. Die Entwicklung der dafür erforderlichen Constraintsprache wurde in der ersten Förderphase gemeinsam von CHORUS und NEP begonnen, in der zweiten Förderphase wechselte Joachim Niehren von NEP zu CHORUS, das seitdem unter gemeinsamer Projektleitung steht. Die Entwicklungen der zweiten CHORUS-Phase wäre ohne die enge Kooperation mit NEP nicht denkbar gewesen. Unabhängig vom konzeptionellen Austausch bei der Constraintmodellierung und -technologie wurde in CHORUS Mozart durchgängig als Implementierungsplattform verwendet.

Die in CHORUS immer wieder notwendige System-Unterstützung wurde von NEP dauerhaft

geleistet. So erweiterte NEP zum Beispiel die Mozart-Mengenconstraints auf Mengen mit mehr als 64 Elementen, was erstmals für die CHORUS Anwendungen erforderlich war. Dazu waren neue Datenstrukturen und ein erheblicher Implementierungsaufwand erforderlich, da große Mengen nicht mehr als einfache Bitleisten realisierbar sind.

Kooperation mit dem Teilprojekt C2 (LISA). Auch in LISA kommen jetzt Constraints und Mozart zum Einsatz. Dafür waren insbesondere die erfolgreiche Anwendungen dieser Technologien in den computerlinguistischen Teilprojekten C3 und C4 verantwortlich. In Zusammenarbeit mit NEGRA, CHORUS und NEP wurde ein Löser für Dominanz-Constraints entwickelt, der diese auf Mengenconstraints zurückführt. Dieser Löser wird von LISA für die Behandlung von Diskursproblemen sowie für einen Parser für lexikalisierte Baumbeschreibungsgrammatiken eingesetzt.

2.5 Vergleiche mit Arbeiten außerhalb des Forschungsbereiches und Reaktionen der wissenschaftlichen Öffentlichkeit auf die eigenen Arbeiten

Das für Oz entwickelte Konzept der programmierbaren Suche gilt mittlerweile als richtungsweisend und wird zunehmend in andere Constraintprogrammiersysteme integriert. Da diese jedoch auf Backtracking und Trailing festgelegt sind (also Prolog-Technologie), gelingt das nur im beschränkten Maße. Beispielsweise ist Suche im ILOG Solver neuerdings programmierbar (ILOG S.A., 2000; Perron, 1999), basiert aber auf einem einfachen Prioritätsmodell, das im Gegensatz zu Oz keine adaptive und interaktive Exploration zulässt (Schulte, 2000b). Eine Variante des Solve-Kombinators (Schulte & Smolka, 1994), ein weniger expressiver Vorläufer von Berechnungsräumen in Oz (es können weder adaptive und interaktive Suche, noch Kombinatoren programmiert werden (Schulte, 2000b)), wurde in die Sprache Curry übernommen (Hanus & Steiner, 1998).

Der Mozart Explorer, ein graphisches und interaktives Werkzeug für Suche in der Constraintprogrammierung, hat mehrere andere Werkzeuge inspiriert (Deransart, Hermenegildo & Małuszyński, 2000), insbesondere den CHIP Search-Tree Visualizer (Simonis & Aggoun, 2000). Diese erreichen jedoch bei weitem noch nicht die Qualitäten des Originals, insbesondere in Hinblick auf Interaktivität und Skalierbarkeit. Der Hauptgrund dafür liegt wie bei der programmierbaren Suche in der Prolog-Technologie dieser Systeme, die hier an ihre Grenzen stößt (Schulte, 2000b).

Bei Sprachen für verteilte Programmierung gibt es zwei interessante neue Entwicklungen, die

sich in ihrer Konzeption von Mozart unterscheiden. Der von Cardelli und Gordon entwickelten den Ambient-Kalkül hat als Grundkonzept so genannte Ambients, die man sich unter anderem als mobile Agenten vorstellen kann, die zwischen Prozessen wandern (die wieder als Ambients modelliert werden) (Cardelli & Gordon, 1998; Gordon & Cardelli, 1999). Mit Ambients können Sicherheitsaspekte von Netzwerken, wie zum Beispiel Firewalls, modelliert werden. Für den Ambient-Kalkül wurde ein Typsystem entwickelt, das Prozesseigenschaften wie Mobilität und Gruppierung beschreiben kann. (Cardelli & Gordon, 1999; Cardelli, Ghelli & Gordon, 1999, 2000).

Eine zweite Entwicklung sind Oderskys funktionale Netze (Odersky, 2000), die funktionale Programmierung mit dem Synchronisationsprinzip von Petri-Netzen kombinieren und Ideen des Join-Kalküls (Fournet, Gonthier, Lévy, Maranget & Rémy, 1996) aufgreifen. Aufbauend auf diesen Ideen entwickelt Odersky die Sprache Funnel. Das von INRIA entwickelte JoCaml (Fournet, Maranget & Schmitt, 2000) erweitert den ML-Dialekt OCaml (Leroy, 2000) um die Kommunikationsprimitive des Join-Kalküls. Im Gegensatz zum π-Kalkül macht der Join-Kalkül Verteilung explizit.

Mozart (Mozart Consortium, 1999a) hat im Berichtszeitraum weite Verbreitung gefunden. Die im Januar 1999 freigegebene Version 1.0 wurde mehr als 7000 mal heruntergeladen, die im Februar 2000 freigegebene Nachfolgerversion 1.1 mehr als 6000 mal. Das System hat eine aktive Benutzergemeinde: an Diskussionen um Sprache und System beteiligen sich mehr als 200 Benutzer, das Nachrichtenaufkommen auf den Mailinglisten liegt bei mehreren Nachrichten am Tag.

2.6 Offene Fragen

Grundlegendere offene Fragen gibt es vor allem in Kontext von Alice. Es ist noch nicht klar, ob die bisher vorgesehen Typstrukturen ausreichen, um alle wichtigen Programmiertechniken aus Mozart in hinreichend flexibler Weise zu unterstützen. Mozart hat beispielsweise ein sehr flexibles Objektsystem, das es in Alice nicht geben wird, da es die Sprache zu komplex und konzeptuell zu redundant machen würde. Auch sonst darf man nicht erwarten, dass sich Programmiertechniken aus einer dynamisch getypten Sprache direkt in eine statisch getypte Sprache übertragen lassen. Stattdessen müssen oft neue Abstraktionen gefunden werden, die zur Typstruktur passen. Für die meisten Kernkonzepte wie logische Variablen, Pickling, Interprozesskommunikation, ganzzahlige Constraints und Berechnungsräume konnten bereits elegante Lösungen gefunden werden. Zu klären bleibt vor allem noch die Typisierung der

Baumconstraints und der Grafikschnittstelle.

Alice soll bis auf eine schlanke virtuelle Maschine in Alice selbst realisiert werden. Für Oz war diese Vorgehensweise sehr erfolgreich. Insbesondere sind der Übersetzer, der Interpreter, der Komponentenmanager und der statische Linker komplett in Oz geschrieben. Für Alice muss geklärt werden, wie die entsprechenden reflektiven Schnittstellen zur virtuellen Maschine typisiert werden können. Bei der sich zur Zeit in Entwicklung befindlichen Alice-Implementierung sind einige der kritischen tiefen Schichten noch mit Oz realisiert (das ist einfach, da wir zur Zeit die virtuelle Maschine von Mozart benutzen).

Im Zusammenhang mit den Operationen für Persistenz und Verteilung können Fehlersituationen auftreten, die durch das Alice-Typsystem nicht ausgeschlossen werden. Erweiterungen des Typsystems, die diese Fehler statisch ausschließen, wären wünschenswert.

Vorarbeiten im laufenden Projektzeitraum haben gezeigt, dass eine Integration des Typklassen-Konzeptes aus Haskell in SML prinzipiell möglich ist (Schneider, 2000). Sie haben aber noch nicht zu einem befriedigenden (hinreichend einfachen) Design geführt. Mit Typklassen lassen sich parametrisierte Typabstraktionen in vielen Fällen einfacher als mit Funktoren realisieren.

Ein wichtiges neues Konzept von Alice ist die dynamische Typprüfung bei der Konversion von Paketen zu Modulen. Es wäre wünschenswert, eine Idealisierung dieser Konversion zu formalisieren und entsprechende Korrektheitseigenschaften zu beweisen.

Literatur

Aiken, A., Kozen, D., Vardi, M. & Wimmers, E. (1993). The Complexity of Set Constraints. In E. Börger, Y. Gurevich & K. Meinke (Hrsg.), *Proceedings of the 7^{th} Conference on Computer Science Logic* (Bd. 832, S. 1–17). Swansea, Wales: Springer-Verlag.

Aiken, A., Kozen, D. & Wimmers, E. (1995). Decidability of systems of set constraints with negative constraints. *Information and Computation, 122*(1), 30–44.

Brock, S. & Ostheimer, G. (1995). Process semantics of graph reduction. In 6^{th} *International Conference on Concurrency Theory* (S. 238–252).

Brunklaus, T. (2000). *Der Oz Inspector - Browsen: Interaktiver, einfacher, effizienter.* Diplomarbeit, Fachbereich 14 Informatik, Universität des Saarlandes.

Cardelli, L., Ghelli, G. & Gordon, A. D. (1999). Mobility types for mobile ambients. In J. Wiedermann, P. van Emde Boas & M. Nielsen (Hrsg.), *Proceedings of the Automata, Languagese and Programming, 26th International Colloquium, ICALP'99* (Bd. 1644, S. 230–239). Springer-Verlag.

Cardelli, L., Ghelli, G. & Gordon, A. D. (2000). Ambient groups and mobility types. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses & T. Ito (Hrsg.), *Proceedings of Theoretical Computer Science; Exploring New Frontiers in Theoretical Informatics. Inter-*

national Conference IFIP TCS 2000 (Bd. 1872, S. 333–347). Sendai, Japan: Springer-Verlag.

Cardelli, L. & Gordon, A. D. (1998). Mobile ambients. In M. Nivat (Hrsg.), *Foundations of Software Science and Computational Structures* (Bd. 1378, S. 140–155). Springer-Verlag.

Cardelli, L. & Gordon, A. D. (1999). Types for mobile ambients. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages, POPL'99* (S. 79–92).

Charatonik, W. & Podelski, A. (1996). The independence property of a class of set constraints. In E. C. Freuder (Hrsg.), *Proceedings of the 2^{nd} International Conference on Principles and Practice of Constraint Programming* (Bd. 1118, S. 76–90). Springer-Verlag.

Deransart, P., Hermenegildo, M. V. & Małuszyński, J. (Hrsg.). (2000). *Analysis and visualization tools for constraint programming: Constraint debugging* (Bd. 1870). Berlin, Germany: Springer-Verlag.

Dörre, J. (1991). Feature logics with weak subsumption constraints. In *Annual Meeting of the ACL (Association of Computational Logics)* (S. 256-263).

Duchier, D. (2000). *MOGUL: The MOzart Global User Library* [Mozart Documentation Series]. http://www.mozart-oz.org/mogul.

Duchier, D., Kornstaedt, L., Schulte, C. & Smolka, G. (1998). *A higher-order module discipline with separate compilation, dynamic linking, and pickling* (Tech. Rep.). http://www.ps.uni-sb.de/papers: Programming Systems Lab, Universität des Saarlandes. (Draft)

Eifrig, J., Smith, S. & Trifonow, V. (1995). Sound polymorphic type inference for objects. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications.* Boston, Massachusetts.

Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L. & Rémy, D. (1996). A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)* (S. 406–421). Pisa, Italy: Springer-Verlag. (LNCS 1119)

Fournet, C., Maranget, L. & Schmitt, A. (2000). *The JoCaml language.* http://pauillac.inria.fr/jocaml/htmlman/.

Gilleron, R., Tison, S. & Tommasi, M. (1993). Solving systems of set constraints with negated subset relationships. In *Proceedings of the 34^{th} IEEE Symposium on Foundations of Computer Science* (S. 372–380). IEEE Computer Society Press.

Gordon, A. D. & Cardelli, L. (1999). Equational properties of mobile ambients. In W. Thomas (Hrsg.), *Proceedings of the Joint European Conferences on Theory and Practice of Software, ETAPS'99* (Bd. 1578, S. 212–226). Amsterdam, The Netherlands: Springer-Verlag.

Hanus, M. & Steiner, F. (1998). Controlling search in declarative programs. In C. Palamidessi, H. Glaser & K. Meinke (Hrsg.), *Principles of Declarative Programming* (Bd. 1490, S. 374–390). Pisa, Italy: Springer-Verlag.

Haridi, S., Van Roy, P., Brand, P., Mehl, M., Scheidhauer, R. & Smolka, G. (1999). Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, *21*(3), 569–626.

Haridi, S., Van Roy, P., Brand, P. & Schulte, C. (1998). Programming languages for distributed applications. *New Generation Computing*, *16*(3), 223–261.

Henglein, F. & Rehof, J. (1997). The complexity of subtype entailment for simple types. In

Proceedings of the 12th IEEE Symposium on Logic in Computer Science (S. 362–372).

ILOG S.A. (2000, August). *ILOG Solver 5.0: Reference manual.* Gentilly, France.

Jaffar, J. & Maher, M. M. (1994). Constraint logic programming: A survey. *The Journal of Logic Programming, 19 & 20,* 503–582. (Special Issue: Ten Years of Logic Programming)

Janson, S. (1994). *AKL - A multiparadigm programming language.* Dissertation, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden. (SICS Dissertation Series 14)

Leroy, X. (2000). *The Objective Caml system release 3.00.* http://pauillac.inria.fr/ocaml/htmlman/.

Marriott, K. & Stuckey, P. J. (1998). *Programming with constraints. an introduction.* Cambridge, MA, USA: The MIT Press.

Mehl, M. (1999). *The Oz Virtual Machine: Records, Transients, and Deep Guards.* Doctoral dissertation, Universität des Saarlandes, Im Stadtwald, 66041 Saarbrücken, Germany.

Melis, E., Zimmer, J. & Müller, T. (2000a). Extensions of constraint solving for proof planning. In W. Horn (Hrsg.), *Proceedings of the 14th European Conference on Artificial Intelligence* (S. 229–233). Berlin: IOS Press.

Melis, E., Zimmer, J. & Müller, T. (2000b). Integrating constraint solving into proof planning. In H. Kirchner & C. Ringeissen (Hrsg.), *Frontiers of Combining Systems – Third International Workshop, FroCos 2000* (Bd. 1794, S. 32–46). Nancy, France: Springer-Verlag.

Milner, R. (1992). Functions as processes. *Journal of Mathematical Structures in Computer Science, 2*(2), 119–141.

Milner, R., Parrow, J. & Walker, D. (1992). A calculus of mobile processes. *Journal of Information and Computation, 100,* 1–77.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1997). *Definition of Standard ML (revised).* Cambridge, MA, USA: The MIT Press.

Müller, M. & Niehren, J. (2000). Ordering constraints over feature trees expressed in second-order monadic logic. *Information and Computation*. (Special Issue on RTA, Tsukuba, Japan, March 1998. To appear)

Müller, M., Niehren, J. & Podelski, A. (2000). Ordering constraints over feature trees. *Constraints, an International Journal, 5*(1–2), 7–42. (Special Issue on CP'97, Linz, Austria.)

Müller, T. & Würtz, J. (1999). Embedding propagators in a concurrent constraint language. *The Journal of Functional and Logic Programming, 1999*(1), Article 8. (Special Issue. Available at: `mitpress.mit.edu/JFLP/`)

Mozart Consortium. (1999a). *The Mozart programming system.* (Available from http://www.mozart-oz.org)

Mozart Consortium. (1999b). *Mozart research groups.* (http://www.mozart-oz.org/people.html)

Müller, M. (1998). *Set-based failure diagnosis for concurrent constraint programming.* Dissertation, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany.

Müller, M. & Niehren, J. (1998). Ordering constraints over feature trees expressed in second-order monadic logic. In T. Nipkow (Hrsg.), *International Conference on Rewriting Techniques and Applications* (Bd. 1379, S. 196–210). Tsukuba, Japan: Springer-Verlag, Berlin.

Müller, M., Niehren, J. & Treinen, R. (1998). The first-order theory of ordering constraints over feature trees. In *Thirteenth annual IEEE Symposium on Logic in Computer Sience (LICS98)* (S. 432–443). Indianapolis, Indiana: IEEE Press. (An extended version is available at http://www.ps.uni-sb.de/Papers/abstracts/FTSubTheory-Long:99.html)

Müller, M. & Nishimura, S. (1998). Type inference for first-class messages with feature constraints. In J. Hsiang & A. Ohori (Hrsg.), *Asian Computer Science Conference (ASIAN 98)* (Bd. 1538, S. 169–187). Manila, The Philippines: Springer-Verlag.

Müller, M. & Nishimura, S. (2000). Type inference for first-class messages with feature constraints. *International Journal of Foundations of Computer Science, 11*(1), 29-63. (Special Issue on ASIAN 98)

Müller, T. (2000a). Practical investigation of constraints with graph views. In R. Dechter (Hrsg.), *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming – CP 2000* (Bd. 1984, S. 320–336). Singapore: Springer-Verlag.

Müller, T. (2000b). Promoting constraints to first-class status. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv & P. J. Stuckey (Hrsg.), *Proceedings of the First International Conference on Computational Logic – CL2000* (Bd. 1861, S. 429–447). London, UK: Springer-Verlag.

Ng, K. B., Choi, C. W., Henz, M. & Müller, T. (2000). GIFT: a generic interface for reusing filtering algorithms. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron & C. Schulte (Hrsg.), *Proceedings of the Workshop on Techniques for Implementing Constraint Programming Systems - TRICS* (S. 86–100). Singapore.

Niehren, J. (2000). Uniform confluence in concurrent computation. *Journal of Functional Programming, 10*(3), 1–47. (An unabridged version from June 30, 1999 is available at http://www.ps.uni-sb.de/Papers/abstracts/Uniform:99.html.)

Niehren, J., Müller, M. & Talbot, J.-M. (1999). Entailment of atomic set constraints is PSPACE-complete. In *Fourteenth annual IEEE Symposium on Logic in Computer Sience (LICS99)* (S. 285–294). Trento, Italy: IEEE Press. (An extended version is available at http://www.ps.uni-sb.de/Papers/abstracts/atomic:98.html)

Niehren, J. & Priesnitz, T. (1999). Entailment of non-structural subtype constraints. In *Asian Computing Science Conference* (S. 251–265). Phuket, Thailand: Springer-Verlag.

Niehren, J. & Priesnitz, T. (2001). *Characterizing non-structural subtype entailment in automata theory* (Tech. Rep.). Universität des Saarlandes, Programming Systems Lab. (Submitted. Available at http://www.ps.uni-sb.de/Papers/abstracts/pauto.html)

Odersky, M. (2000). Functional nets. In G. Smolka (Hrsg.), *Proceedings of the 9th European Symposium on Programming, ESOP 2000* (Bd. 1782). Berlin, Germany: Springer-Verlag.

Perron, L. (1999). Search procedures and parallelism in constraint programming. In J. Jaffar (Hrsg.), *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming* (Bd. 1713, S. 346–360). Alexandra, VA, USA: Springer-Verlag.

Podelski, A., Charatonik, W. & Müller, M. (1999). Set-based failure analysis for logic programs and concurrent constraint programs. In S. D. Swierstra (Hrsg.), *Proceedings of ESOP'99, the European Symposium of Programmin.* Amsterdam: Springer-Verlag.

Pollard, C. J. & Sag, I. A. (1987). *Information-based syntax and semantics, vol. 1.* Stanford University: Center for the Study of Language and Information. (Distributed by University of Chicago Press)

Pottier, F. (1996). Simplifying subtyping constraints. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming* (S. 122–133). ACM Press, New York.

Schneider, G. (2000). *ML mit Typklassen.* Diplomarbeit, Fachrichtung Informatik, Universität des Saarlandes.

Schulte, C. (1997a). Programming constraint inference engines. In G. Smolka (Hrsg.), *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming* (Bd. 1330, S. 519–533). Schloß Hagenberg, Linz, Austria: Springer-Verlag.

Schulte, C. (1997b). Oz Explorer: A visual constraint programming tool. In L. Naish (Hrsg.), *Proceedings of the Fourteenth International Conference on Logic Programming* (S. 286–300). Leuven, Belgium: The MIT Press.

Schulte, C. (1999). Comparing trailing and copying for constraint programming. In D. De Schreye (Hrsg.), *Proceedings of the 1999 International Conference on Logic Programming* (S. 275–289). Las Cruces, NM, USA: The MIT Press.

Schulte, C. (2000a). Parallel search made simple [Technical Report]. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron & C. Schulte (Hrsg.), *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000* (S. 41–57). 55 Science Drive 2, Singapore 117599.

Schulte, C. (2000b). *Programming constraint services.* Dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany. (To appear in LNAI, Springer-Verlag)

Schulte, C. (2000c). Programming deep concurrent constraint combinators. In E. Pontelli & V. S. Costa (Hrsg.), *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000* (Bd. 1753, S. 215–229). Boston, MA, USA: Springer-Verlag.

Schulte, C. & Smolka, G. (1994). Encapsulated search in higher-order concurrent constraint programming. In M. Bruynooghe (Hrsg.), *Logic Programming: Proceedings of the 1994 International Symposium* (S. 505–520). Ithaca, NY, USA: The MIT Press.

Simonis, H. & Aggoun, A. (2000). Search-tree visualization. In P. Deransart, M. V. Hermenegildo & J. Małuszyński (Hrsg.), *Analysis and visualization tools for constraint programming: Constraint debugging* (Bd. 1870, S. 191–208). Berlin, Germany: Springer-Verlag.

Smolka, G. (1994). A foundation for concurrent constraint programming. In *Constraints in Computational Logics* (Bd. 845, S. 50–72). Springer-Verlag.

Smolka, G. (1995a). The definition of Kernel Oz. In A. Podelski (Hrsg.), *Constraint programming: Basics and trends* (Bd. 910, S. 251–292). Springer-Verlag.

Smolka, G. (1995b). The Oz programming model. In J. van Leeuwen (Hrsg.), *Computer science today* (Bd. 1000, S. 324–343). Berlin: Springer-Verlag.

Smolka, G. (1998). Concurrent constraint programming based on functional programming. In C. Hankin (Hrsg.), *Programming Languages and Systems* (Bd. 1381, S. 1–11). Lisbon, Portugal: Springer-Verlag.

Smolka, G. (1999). *From concurrent constraint programming to concurrent functional programming with transients* [Slides]. Summer School on Constraints in Computational Logics, Gif-sur-Yvette, France. (http://www.ps.uni-sb.de/ smolka/ccl99.ps)

Van Hentenryck, P., Saraswat, V. et al.. (1997). Strategic directions in constraint programming. *ACM Computing Surveys, 28*(4), 701–726. (ACM 50th Anniversary Issue. Stra-

tegic Directions in Computing Research.)

Van Hentenryck, P., Simonis, H. & Dincbas, M. (1992). Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, *58*, 113–159.

Wallace, M. (1996). Practical applications of Constraint Programming. *Constraints*, *1*(1/2), 139–168.

Zimmer, J. (2000, May). *Constraintlösen für Beweisplanung*. Diplomarbeit, Fakultät für Mathematik und Informatik, Fachbereich Informatik, Universität des Saarlandes. (In German)