Teilprojekt MI 6:

NEP:
Statisch getypte Programmierumgebung für nebenläufige Constraints

3.1 Allgemeine Angaben zum Teilprojekt MI 6

3.1.1 Thema

NEP: Statisch getypte Programmierumgebung für nebenläufige Constraints

3.1.2 Fachgebiet und Arbeitsrichtung

Informatik, Programmiersysteme

3.1.3 Leiter

Name, Geburtsdatum:	Prof. Dr. Gert Smolka, 05.01.1955
Dienstanschrift:	FR Informatik Universität des Saarlandes Postfach 15 11 00 66041 Saarbrücken
Telefon:	(0681) 302 5311
Telefax:	(0681) 302 5341
E-mail:	smolka@ps.uni-sb.de

Ist die Stelle des Leiters des Projektes befristet?

[X] Nein [ ] Ja, befristet bis zum

3.1.4 Überführung

Entfällt.

3.1.5 Vorgesehene Untersuchungen

In dem Teilprojekt sind vorgesehen:

Untersuchungen am Menschen	[ ] ja	[X] nein
klinische Studien im Bereich der somatischen Zell- oder Gentherapie	[ ] ja	[X] nein
Tierversuche	[ ] ja	[X] nein
gentechnologische Untersuchungen	[ ] ja	[X] nein

3.1.6 Bisherige und beantragte Förderung des Teilprojektes im Rahmen des Sonderforschungsbereichs (*Ergänzungsausstattung*)

Haushaltsjahr	Personalkosten	Sächliche Verw.-ausgaben	Investitionen	gesamt
bis 1998	1015.2	–	–	1015.2
1999	240	–	–	240
2000	240	–	–	240
2001	240	–	–	240
Zwischensumme	1735.2	–	–	1735.2
2002	254.4	–	–	254.4
2003	254.4	–	–	254.4
2004	254.4	–	–	254.4

(Beträge in TDM)

3.2 Zusammenfassung

Das Teilprojekt soll das Programmiersystem Alice zur Anwendungstauglichkeit weiterentwickeln. Alice stellt die aus Mozart bekannte Funktionalität (Futures, Constraints, Persistenz, Verteilung) in vereinfachter und verbesserter Form zur Verfügung. Eine wesentliche Neuerung von Alice besteht darin, dass es die Mozart-Funktionalität in einem statisch getypten Rahmen zur Verfügung zu stellt. Dadurch kann die konsistente Verwendung von Schnittstellen und Abstraktionen während der Programmentwicklung automatisch überprüft werden. Diese Konsistenzprüfungen reduzieren den Aufwand des Anwenders bei der Entwicklung und Erweiterung von Software. Programmiersprachlich ist Alice eine Erweiterung der funktionalen Programmiersprache Standard ML.

Aus Forschungssicht interessieren uns insbesondere zwei Fragen:

1. Wie lassen sich Constraint- und Internet-Programmierung in einem statisch getypten Rahmen elegant modellieren? Welche zusätzlichen Typisierungskonzepte sind notwendig? Diese Fragen sollen auch in Kooperation mit den Teilprojekten geklärt werden, die diese Programmierkonzepte im Rahmen von Mozart benutzen.
2. Wie kann ein Programmiersystem für Alice möglichst einfach, modular und effizient realisiert werden? Was sind die Prinzipien für eine plattformunabhängige Realisierung mithilfe einer virtuellen Maschine? Wie kann die Maschine konzeptuell in allgemeine und wiederverwendbare Komponenten zerlegt werden?

Wie bisher soll NEP andere Teilprojekte bei der Anwendung von Mozart unterstützen. Gegebenenfalls kann fehlende Constraintfunktionalität für die Anwender nachgerüstet werden. Da der existierende Alice-Prototyp mithilfe der Mozart-VM realisiert ist, können Alice- und Mozart-Programme gemischt werden. Das erlaubt Anwendern einen schrittweisen Übergang von Mozart zu Alice.

3.3 Stand der Forschung

3.3.1 Programmiersysteme mit zu Mozart/Alice vergleichbarer Funktionalität

Die durch Mozart insgesamt realisierte Funktionalität ist einmalig und wird durch kein anderes Programmiersystem angeboten. Sinnvolle Vergleiche sind jedoch für die Teilfunktionalitäten Constraints und verteilte Programmierung möglich. Außerdem ist ein Vergleich mit existierenden ML-Systemen sinnvoll.

Vergleich mit Constraintprogrammiersystemen. Es gibt mehrere Constraintprogrammiersysteme (ILOG Solver, Sicstus Prolog, Gnu Prolog), die für Finite Domain Constraints ähnliche Funktionalität wie Mozart anbieten. Was Abdeckung und Effizienz anbetrifft, ist Mozart mindestens so gut, wie die besten dieser Systeme. Mozart unterscheidet sich jedoch von diesen Systemen grundlegend dadurch, wie es Suche realisiert. Hier arbeitet Mozart mit nebenläufigen Berechnungsräumen, die mit Kopieren statt mit Backtracking und Trailing realisiert werden. Mithilfe von Berechnungsräumen lassen sich Suchmaschinen programmieren, die mit Backtracking prinzipiell nicht realisierbar sind. Ein Beispiel ist der interaktive Oz Explorer. Außerdem lassen sich mit Berechnungsräumen kompositionale Constraintkombinatoren programmieren, die ebenfalls in keinem anderen System existieren (aber in Mozart-Anwendungen eine wichtige Rolle spielen). Schließlich ist Mozart das einzige Programmiersystem, das über eine verteilte Suchmaschine verfügt. Mozart übertrifft also alle anderen

Constraintprogrammiersysteme um eine Größenordnung, was die angebotene Funktionalität angeht. Diese Punkte werden erschöpfend in der Dissertation von Schulte (2000b) diskutiert.

Bis auf die C++ Bibliothek ILOG Solver sind die existierenden Constraintprogrammiersysteme mit dynamisch getypten Sprachen realisiert (fast immer Prolog). Ein mit einer statisch getypten funktionalen Sprache realisiertes Constraintprogrammiersystem gibt es bisher nicht.

Vergleich mit Systemen für verteilte Programmierung. Die zwei wichtigsten Systeme in dieser Gruppe sind Java und Erlang. Ein wesentlicher Unterschied dieser Systeme zu Mozart ist die Tatsache, dass Mozart höherstufige Objekte geschlossen inklusive ihres Codes kommuniziert. Java und Erlang kommunizieren dagegen Code getrennt von Datenstrukturen, was zu einer Reihe von Problemen führt. Anwender von Mozart berichten, dass die Robustheit, Skalierbarkeit und Effizienz der Verteilungsfunktionalität von Mozart die von Erlang und Java bei weitem übertrifft. Mozart bietet zudem mit Futures ein sehr flexibles Mittel, um die mit Kommunikation verbundenen Verzögerungen (Latenz) auszugleichen.

Vergleich mit ML-Systemen. OCaml (Leroy, 2000), SML of New Jersey (Lucent, 2000) und Moscow ML (Romanenko, Russo & Sestoft, 2000b) sind drei ausgereifte ML-Systeme. OCaml ist mittels Vollübersetzung realisiert und erreicht fast die Effizienz von C++. Alice wird weniger effizient sein, dafür aber eine Menge an Funktionalität anbieten, die in keinem dieser Systeme verfügbar ist: Dynamisches Laden von Komponenten, Futures, Pakete und dynamische Typprüfung, persistente Speicherung von dynamischen Modulen, Interprozesskommunikation von dynamischen Modulen, Constraints und Suche.

3.3.2 Virtuelle Maschinen

Virtuelle Maschinen werden für die Implementierung logischer (Warren, 1983; AitKaci, 1991), funktionaler (zum Beispiel OCaml (Leroy, 1990)) und objektorientierter (zum Beispiel Java (Lindholm & Yellin, 1997)) Programmiersprachen eingesetzt. Auch die neue .NET-Plattform von Microsoft (Microsoft, 2000) ist mithilfe einer virtuellen Maschine definiert. Bisher werden virtuellen Maschinen im Wesentlichen als monolithische Interpreter beschrieben, die einen Instruktionssatz ausführen. Diese Sicht ist naiv, da alle virtuellen Maschinen neben der Ausführung von Instruktionen andere Aufgaben erledigen, die von erheblicher Komplexität sind. Die JVM ist ein gutes Beispiel: sie lädt Classfiles, kontrolliert Threads und integriert oft einen Laufzeitübersetzer (Just in time compilation).

Alle bekannten Ansätze gehen von einem einzigen Interpreter aus. Es wird keine Unterschei-

dung zwischen abstraktem und internen Instruktionssatz gemacht. Der Instruktionssatz kombiniert damit sprachspezifische mit architekturspezifischen Aspekten. Damit sind diese Maschinen weitgehend auf eine bestimmte Implementierung festgelegt. Beispielsweise ist bei allen oben angeführten virtuellen Maschinen der Instruktionssatz sehr reich und daraufhin optimiert, auf Desktop Computern möglichst effizient zu sein. Das schließt die Verwendung der virtuellen Maschine für mobile Geräte aus.

Laufzeitübersetzung wird bisher nur für nativen Zielcode betrachtet. Die bei der Integration von Laufzeitübersetzung in die JVM auftretenden Schwierigkeiten werden in (Ogawa et al., 2000) beschrieben. Bisher ist nicht untersucht, wie ein Instruktionssatz so entworfen werden kann, dass er besonders gut für Laufzeitübersetzung (insbesondere in verschiedene interne Instruktionssätze) geeignet ist. Außerdem fehlt eine Architektur für die Integration von Laufzeitübersetzung in eine virtuelle Maschine. Interne Instruktionssätze könnten auf Effizienz oder auf explizite Darstellung von Ablaufinformation für Debugging-Zwecke ausgerichtet sein.

Bei der Speicherverwaltung sind bereits seit längerer Zeit wiederverwendbare Komponenten verfügbar. Mali (Bekkers, Canet, Ridoux & Ungaro, 1986) wurde für die Speicherverwaltung von logischen Programmiersprachen entwickelt. Weit verbreitet sind konservative Speicherbereiniger (Boehm & Weiser, 1988; Wilson, 1992; Jones & Lins, 1996), die ohne weitere Unterstützung auskommen, dafür aber eventuell mehr Speicher als nötig verwenden.

3.3.3 Dynamische Typisierung

Die Einbettung dynamischer Typisierung in ein statisches Typsystem wurde erstmals von Abadi, Cardelli, Pierce und Plotkin (1989) formalisiert. Sie erweitern den einfach getypten λ-Kalkül um einen Typ Dynamic, der ein Paar aus einem beliebigen Wert und dessen Typ enkapsuliert. Ein solches Objekt kann mit einem speziellen Typecase-Konstrukt inspiziert und der enthaltene Wert projiziert werden. Ihr Typecase ist expressiv und erlaubt das Pattern-Matching auf beliebige Typmuster mit Variablen. In folgenden Arbeiten übertragen sie ihren Ansatz auf den polymorphen λ-Kalkül (Abadi, Cardelli, Pierce & Rémy, 1995). Das System erzwingt allerdings, dass der Typ eines in Dynamic eingebetteten Wertes immer geschlossen ist, oder aber dass für alle polymorphen Typvariablen zur Laufzeit Typen übergeben werden, was unerwünschte Kosten zur Folge hat. Einen ähnlichen Ansatz verfolgen Leroy und Mauny (1993). Sie verschmelzen Typecase mit dem normalen Case-Konstrukt. Ihr System ist allerdings weniger expressiv.

Dubois, Rouaix und Weis (1995) stellen eine funktionale Sprache vor, die es polymorphen Funktionen erlaubt, die konkrete Instanziierung ihrer Typparameter mittels eines einfachen Typecase-Konstrukts zu inspizieren und in entsprechende Subfunktionen zu verzweigen. Um eine effiziente Implementierung zu ermöglichen, werden diese so genannten generischen Typvariablen von gewöhnlichen polymorphen Typvariablen unterschieden. Für jede generische Typvariable wird zur Laufzeit implizit eine Typrepräsentation an die Funktion übergeben. Damit ähnelt der Ansatz der Idee des Dictionary-Passings für mittels Typklassen eingeschränkter Typvariablen in Haskell, wobei Typecase an die Stelle von Instanzdeklarationen tritt.

Harper und Morisett formalisieren im λ_i^{ML}-Kalkül einen stärker typtheoretisch inspirierten Ansatz zur Laufzeit-Typanalyse. Ihr Kalkül ist primär als interne Repräsentation für typgerichtete Compiler entworfen, die Typinspektion zur Implementierung spezieller typgerichteter Operationen, wie zum Beispiel markierungsfreie Speicherbereinigung (tag-free garbage collection) benötigen. Ihr Kalkül basiert deshalb auf dem Lambda-Kalkül höherer Ordnung und ist damit explizit getypt. Der Kalkül enthält auf Termebene ein Typecase-Konstruct, mit dem beliebige Typargumente inspiziert werden können, als auch ein Typecase-Konstrukt auf Typebene, was die Expressivität zusätzlich erhöht. Crary, Weirich und Morisett (1998) stellen den λ_i^{ML}-Kalkül vor und geben eine typlöschende Transformation in einen einfacheren Kalkül an, bei der Typen durch repräsentierende Terme ersetzt werden. Dies ist Voraussetzung für eine einfache Implementierung.

Verschiedene Sprachen stellen einen Typ ähnlich Dynamic zur Verfügung. In Mercury (Henderson et al., 2000) beispielsweise ist er unter dem Namen univ verfügbar. Da Mercury grundsätzlich Laufzeit-Typbeschreibungen an polymorphe Prädikate übergibt, gibt es keine Geschlossenheitsbeschränkung. Die Bibliothek des Haskell-Compilers GHC (Marlow, Peyton Jones & Others, 1999) kennt ebenso einen solchen Typ. Die Verwendung von dynamischer Typisierung ist dabei nur für Instanztypen einer speziellen vordefinierten Typklasse erlaubt, so dass die Geschlossenheitsbeschränkung auf elegante Weise zu Gunsten einer spezifischen Typisierung aufgegeben werden kann. Beide Sprachen stellen aber kein Typecase-Konstrukt zur Verfügung, sondern beschränken sich auf dynamisch überprüfte Projektionsoperationen. Zudem erlauben sie nur die Injektion monomorpher Werte.

Alle Arbeiten betrachten im Wesentlichen nur dynamische Typisierung für Objekte der Kernsprache. Es existiert kein konkreter Vorschlag für dynamische Typisierung von Modulen. Auch scheint bisher keine konkrete Programmiersprache zu existieren, die wirklich ein vollwertiges Typecase-Konstrukt zur Verfügung stellt.

3.3.4 Typklassen und qualifizierter Polymorphismus

Typklassen sind ein Sprachkonzept, das für die Programmiersprache Haskell erfunden wurde. Die ursprüngliche Motivation für Typklassen war, die Überladung von Operatoren auf eine systematische Basis zu stellen (Wadler & Blott, 1989; Kaes, 1988), die mit polymorpher Typinferenz verträglich ist. Später wurde deutlich, dass die zu Grunde liegende Idee, parametrischen Polymorphismus um Constraints zu ergänzen, ein wesentlich allgemeineres Ausdrucksmittel darstellt, welches generische Programmieransätze ermöglicht (Jones, 1994).

Jones (1993) führt Polymorphismus höherer Ordnung ein und verallgemeinert Typklassen erster Ordnung zu Klassen über höherstufigen Typkonstruktoren. Konstruktorklassen wurden in das Design von Haskell 98 (Peyton Jones, Hughes et al., 1998) aufgenommen.

Läufer (1996) kombiniert Typklassen mit existentieller Quantifizierung. Konstruktortypen können einen existentiell quantifizierten Argumenttyp tragen, der einen beliebigen Klassenkontext besitzen darf. Dadurch werden heterogene Datenstrukturen und objektorientierte Programmieransätze ermöglicht, ohne dass die Typinferenz beeinträchtigt wird. Jones (1997) erweitert Läufers Arbeit um universelle Quantifizierung. Damit stehen sowohl emanzipierter Polymorphismus als auch emanzipierte abstrakte Typen zur Verfügung. Beides ist in den meisten gängigen Haskell-Systemen als Erweiterung implementiert.

Peyton Jones, Jones und Meijer (1997) diskutieren verschiedene Erweiterungen zu Typklassen, insbesondere die Verallgemeinerung von Klassen als einstellige Prädikate über Typen zu beliebigen mehrstelligen Relationen (Multiparameter-Typklassen). Andere dort diskutierte Erweiterungen steigern zwar ebenfalls die Expressivität des Typsystems, führen jedoch teilweise zu einem Verlust der Entscheidbarkeit der Typüberprüfung. Sowohl der Glasgow Haskell Compiler (Marlow et al., 1999) als auch der Interpreter Hugs (Jones, Reid & Others, 2001) implementieren die meisten Vorschläge als experimentelle Erweiterungen.

Jones (2000) behandelt das Problem häufiger Typambiguitäten im Zusammenhang mit Multiparameter-Typklassen, indem funktionale Abhängigkeiten zwischen den Klassenparametern spezifiziert werden können. Funktionale Abhängigkeiten sind ebenfalls in den neuesten Versionen von Hugs und GHC verfügbar.

Hinze und Peyton Jones (2000) schlagen vor, Haskells Mechanismus zur Spezifikation von Default-Methoden in Klassendeklarationen zu einem vollwertigen Mechanismus für generische (polytypische) Funktionen auszubauen.

Lewis, Shields, Meijer und Launchbury (2000) entwerfen eine funktionale Sprache mit impliziten Argumenten. Die Typisierung folgt über Typconstraints ähnlich denen von Typklassen. Sie argumentieren, dass implizite Argumente als ein Baustein zur Rückführung von Typklassen auf primitivere Mechanismen verstanden werden können.

Auf der theoretischen Seite schuf (Jones, 1994) mit der Theorie von qualifizierten Typen eine formales Erklärungsmodell für Typklassen und verwandte Erweiterungen des Hindley/Milner-Typsystems und zeigt dessen Wohldefiniertheit. Zudem diskutiert er verschiedene Implementierungsaspekte.

Nipkow und Prehofer (1995) untersuchen den Typinferenzalgorithmus für Typklassen und betrachten dabei insbesondere die notwendige Erweiterung der Unifikation um einen Constraint-Löser.

Sulzmann, Odersky und Wehr (1999), Sulzmann (2000) stellen das allgemeine Rahmensystem HM(X) vor, welches Hindley/Milner-Typisierung über ein beliebiges Constraintsystem parametrisiert. Sie zeigen, dass eine Instanz des Systems alle wünschenswerten theoretischen Eigenschaften erfüllt, wenn das Constraintsystem bestimmte minimale Voraussetzungen erfüllt. Typklassen können als eine Instanz von HM(X) verstanden werden.

Die Expressivität von Typklassen wurde in einigen ambitionierten Bibliotheken-Designs exploriert. Beispielsweise diskutiert Peyton Jones (1996), wie der Grad an Polymorphismus bei generischen Container-Datenstrukturen und darauf operierenden Operationen mit Hilfe von Konstruktorklassen maximiert werden kann. Okasaki (2000) stellt einen Entwurf einer Container-Bibliothek vor, der Eigenschaften und Operationen noch weiter orthogonalisiert und dazu starken Gebrauch von Multiparameter-Klassen macht. Sage (2000) entwickelte FranTK, eine rein funktionale Bibliothek für grafische Benutzeroberflächen, die ebenfalls viele Aspekte der Haskell-Typklassen ausreizt, um objektorientierte Idiome ersetzen zu können. Jeffery, Dowd und Somogyi (1999) schließlich stellen eine Anbindung von Mercury an den objektorientierten Schnittstellen-Standard CORBA vor, die ebenfalls objektorientierte Konstrukte auf Typklassen abbildet.

3.3.5 Records und Varianten

Eines der schwierigeren Probleme statischer Typsysteme ist die polymorphe Typisierung von Operationen auf Records (Verbunden oder Produkten). Die meisten Programmiersprachen, darunter auch SML, erzwingen beispielsweise, dass der Typ eines Records x bei der Selek-

tionsoperation $x.a$ statisch vollständig bekannt ist. Dual dazu ist auch die Handhabung von Varianten (disjunkte Summen) schwierig. Den Umgang mit diesen Konstrukten bei statischer Typisierung flexibler zu machen ist das Ziel verschiedener Forschungsarbeiten in jüngerer Zeit.

Rémy und Vouillon (1998) präsentieren ein System von polymorphen Records, welches auf Zeilenvariablen basiert (row polymorphism), und modellieren ein Objektsystem für eine funktionale Sprache mit Typinferenz mit ihrem Ansatz. Die Arbeit setzt ältere Arbeiten von Rémy (1993) zur Recordtypisierung fort. Die Typisierung des Objektsystems von OCaml beruht auf diesem Ansatz (Leroy, 2000).

Ohori (1995) beschreibt einen etwas anderen Ansatz zur polymorphen Typisierung von Recordoperationen, der auf einem involvierten Kindsystem zur Differenzierung der Teilinformation über einen Recordtyp beruht. Sein System hat eine vergleichbare Expressivität wie Rémys Ansatz.

Pottier (1998) untersucht Typinferenz in Typsystemen mit Subtyping auf Grundlage eines Constraint-basierten Systems. Er präsentiert verschiedene Regeln zur Vereinfachung dabei auftretender Teilconstraints. Basierend auf dieser Arbeit entwickelt er ein allgemeines Typinferenzsystem mit Subtyping und Zeilenvariablen, welches verschiedene Recordoperationen (einschließlich Record-Konkatenation), aber auch polymorphe Varianten handhaben kann (Pottier, 2000). Das zu Grunde liegende Constraintsystem ist jedoch sehr komplex und seine Praktikabilität unklar.

Garrigue (1998, 2000) erweitert das ML-Typsystem um polymorphe Varianten. Sein Ansatz basiert auf einer nicht trivialen Erweiterung der Typinferenz mit Zeilenvariablen. Polymorphe Varianten sind in der neuesten Version von OCaml verwirklicht (Leroy, 2000).

Gaster und Jones (1996) entwickelten einen Ansatz zur polymorphen Typisierung von erweiterbaren Records und Varianten basierend auf Jones' Theorie der qualifizierten Typen (Jones, 1994). Teile davon sind im Hugs-Interpreter implementiert (Jones et al., 2001). Jones und Peyton Jones (1999) entwickeln den Record-Teil dieser Arbeit weiter und schlagen die Integration in Haskell vor.

3.3.6 Modulsysteme

Module sind das grundlegende Strukturierungsmittel für größere Programme. Standard ML war die erste Sprache, für die ein theoretisch fundiertes, parametrisches Modulsystem entworfen wurde. Die Fundierung und Erweiterung der SML-Modulsprache ist nach wie vor ein aktives Forschungsfeld.

MacQueen (1986) entwickelte die grundlegende Idee, Module mit Hilfe von abhängigen Typen (dependent types) zu typisieren. Er führte das Konzept eines *Funktors* ein, der Module auf Module abbildet. Die Anwendung eines Funktors ist *generativ*, was bedeutet, dass seine Anwendung jeweils neue Typen erzeugt. MacQueen und Tofte (1994) erweitern das SML-Modulsystem um höherstufige Funktoren.

Harper, Mitchell und Moggi (1990) modellieren die SML-Modulsprache in einer Variante von F_ω, dem Lambda-Kalkül höherer Ordnung, und erhalten damit auch ein formales Modell für höherstufige Funktoren. Sie zeigen zudem die so genannte Phasentrennungseigenschaft, die garantiert, dass die Typüberprüfung von Modulausdrücken trotz abhängiger Typisierung getrennt von der Programmausführung vorgenommen werden kann, also entscheidbar ist.

Darauf aufbauend entwickeln (Harper & Lillibridge, 1994; Lillibridge, 1997) den Translucent-Sum-Kalkül, der eine verallgemeinerte Form von Modulen im Rahmen der Typtheorie fundiert. Der Kalkül identifiziert Kern- und Modulsprache und stellt somit natürlicherweise auch emanzipierte Module (first-class modules) und Signaturen dar. Lillibridge beweist, dass Typprüfung für diesen Kalkül unentscheidbar ist.

Leroy (1995) präsentiert ein ähnliches System, aber mit beibehaltener Stratifizierung zwischen Kern- und Modulsprache. Sein System weist permissiblere Typisierungsregeln für Funktorapplikation auf. Dadurch sind Funktoren aus Sicht des Typsystems applikativ, was eine präzisere Typisierung bestimmter höherstufiger Beispiele ermöglicht. Allerdings sind diese applikativen Typisierungsregeln inkompatibel mit Funktoren, die eine wirklich generative Semantik besitzen. Leroys Arbeit ist Basis des Modulsystems von Objective Caml (Leroy, 2000), welches zusätzlich emanzipierte Signaturen erlaubt und somit ebenfalls unentscheidbar ist. Interessanterweise hat diese Unentscheidbarkeit in der Praxis keine negativen Auswirkungen.

Russo (1999) zeigt, dass sich das SML-Modulsystem auch in einer einfacheren Theorie zweiter Ordnung ohne abhängige Typen ausdrücken lässt. Seine Formalisierung lässt sich relativ direkt auf Funktoren höherer Ordnung sowie emanzipierte Module erweitern (Russo, 2000, 1998). Seine Formalisierung erlaubt sowohl applikative als auch generative Funktoren, aller-

dings kann jeder generative Funktor mittels Eta-Expansion in einen applikativen konvertiert werden, was den Nutzen der generativen Funktoren stark reduziert. In Bezug auf emanzipierte Module bleibt die Stratifizierung zwischen Kern- und Modulsprache erhalten, was zwar die Expressivität einschränkt, jedoch die Entscheidbarkeit der Typüberprüfung beibehält. Moscow ML 2.00 ist die erste ML-Implementierung mit emanzipierten Modulen (Romanenko, Russo & Sestoft, 2000a).

Flatt und Felleisen (1998) entwickeln ein formales Modell für Programmkomponenten, die sie *units* nennen. Ihr Modell beschreibt getrennte Übersetzung, hierarchische Modularisierung und dynamisches Laden.

Sewell (2001) untersucht das Problem der Versionierung von Software-Komponenten in einem stark verteilten Kontext wie dem Internet. Er stellt einen statisch typisierten Kalkül vor, der dem Entwickler detailierte Kontrolle über verschiedene Versionen von Modulen und abstrakten Typen gibt.

3.3.7 Typen und Effekte

Gifford und Lucassen (1986) schlagen vor, statische Typsysteme um so genannte Effektsysteme zu erweitern, die Seiteneffekte von Berechnungen in Form von Typannotationen erfassen. Nielson und Nielson (1999) geben einen Überblick über den aktuellen Stand der Forschung. Im Kontext von ML seien vor allem die Arbeiten von Talpin und Jouvelot erwähnt, die mit Facile eine konkrete, auf SML basierende Sprache mit Effektinferenz entwickelten (siehe zum Beispiel Talpin & Jouvelot, 1994), sowie das Region-Inferenzsystem von Tofte und Birkedal (1998).

Wadler (1992) führte die Idee ein, imperative Berechnungen in Monaden zu enkapsulieren und so in funktionale Sprachen zu integrieren, ohne die Eigenschaft der referentiellen Transparenz zu zerstören. So lässt sich insbesondere Ein-/Ausgabe in einem rein funktionalen Kontext realisieren (Peyton Jones & Wadler, 1993), wie in der Sprache Haskell verwirklicht. In einer jüngeren Arbeit vergleicht Wadler monadische Typisierung mit Effektsystemen und zeigt, dass beide im Wesentlichen die gleiche Ausdruckskraft besitzen (Wadler, 1998).

3.4 Eigene Vorarbeiten

Übersetzer und virtuelle Maschinen. In diesem Bereich haben wir bei der Entwicklung von Mozart reichlich Erfahrung gesammelt. Eine Übersicht über die virtuelle Maschine für Mozart geben Mehl, Scheidhauer und Schulte (1995). Scheidhauer (1998) beschreibt die Basiskomponenten der virtuellen Maschine und grundlegenden Übersetzungstechniken. Der Schwerpunkt von Mehl (1999) liegt auf der Beschreibung von Implementierungstechniken für logische Variablen und Futures. Die Implementierung von Berechnungsräumen wird von Schulte (2000b) beschrieben.

Eine wichtige Vorarbeit für die Realisierung des Laufzeitsystems für Alice ist das Laufzeitsystem von Mozart. Dazu gehören Systembibliotheken und ein Komponentenmanager (Duchier, Kornstaedt, Schulte & Smolka, 1998).

Vorarbeiten für die Verteilung sind die für Mozart entwickelten Verteilungsprotokolle (Van Roy et al., 1997; Haridi et al., 1999), generelle Techniken für die Verteilung (Haridi, Van Roy, Brand & Schulte, 1998) und der Network Layer von Mozart.

Bibliotheken und Werkzeuge. Neben der Erfahrung bei der Entwicklung von Bibliotheken für Mozart sind insbesondere die Vorarbeiten für Bibliotheken für Constraints (Müller & Würtz, 1999; Würtz, 1998), Suche und Kombinatoren (Schulte, 1997a, 1999, 2000c, 2000a, 2000b) zu erwähnen. Vorarbeiten für die Entwicklung von Werkzeugen sind der Oz Explorer als visuelles Werkzeug für Suche (Schulte, 1997b), ein visuelles Werkzeug für Constraints (Müller, 2000), und der Inspector für die Darstellung von Laufzeitobjekten (Brunklaus, 2000).

Alice. In der laufenden Förderphase haben wir mit dem Entwurf und der Implementierung des Alice-Systems begonnen (Alice Team, 2001). Ein ausführliche Darstellung dieser Vorarbeiten befindet sich im Berichtsband.

In einem Projekt mit Microsoft Research und anderen Forschungsgruppen waren wir an der konzeptionellen Validierung der virtuellen Maschine der .NET-Plattform (Microsoft, 2000) beteiligt. Im Rahmen dieses Projekts haben wir für .NET einen prototypischen Übersetzer für Alice entwickelt.

3.5 Arbeitsprogramm (Ziele, Methoden, Zeitplan)

Mozart realisiert in mehrerer Bereichen Funktionalität, die in anderen Programmiersystemen nicht verfügbar ist. Diese Funktionalität wird von einer wachsenden Benutzergemeinde nachgefragt. Im SFB arbeiten alle C-Projekte und B1 mit Mozart. Mozart wurde nicht nach einem Masterplan gebaut, sondern schrittweise um die Ergebnisse von explorativen Forschungsprojekten erweitert. Diese evolutionäre Entwicklung der Programmiersprache und des Systems verlangte viele Kompromisse, hatte aber den Vorteil, dass die neue Funktionalität von vielen Benutzer an vielen Anwendungen erprobt werden konnte.

Mozart ist also wie eine Gebäude, dass fortlaufend an neue Erfordernisse angepasst und an sehr vielen Stellen durch ganz verschiedene Anbauten erweitert wurde. Der Zeitpunkt für den Bau eines neuen Gebäudes ist gekommen. Dieses kann viel einfacher als das alte gestaltet werden, da die Erfordernisse jetzt von vorneherein klar sind. Das neue Gebäude ist einfacher zu benutzen (wichtig für die Anwender), einfacher zu betreiben (wichtig für die Entwickler) und bei neuen Anwendungen einfacher zu erweitern.

Der Mozart-Nachfolger heißt Alice und seine Plannung und Realisierung wurden in der laufenden Förderperiode begonnen. Anders als Mozart ist Alice als konservative Erweiterung einer etablierten Programmiersprache konzipiert. Damit sind die innovativen Aspekte sauber von den etablierten Aspekten getrennt, was die Benutzung sehr vereinfacht. Als Kernsprache haben wir Standard ML (SML) gewählt, eine Sprache, die in der programmiersprachlichen Forschung eine prominente Stellung einnimmt und die wir in Saarbrücken seit 4 Jahren mit großem Erfolg in der Anfängervorlesung verwenden.

SML ist eine funktionale Sprache. Damit wird ein initialer Designfehler von Mozart behoben, der sich aus Mozarts Kindheit in der logischen Programmierung erklärt. SML ist auch eine statisch getypte Sprache, mit einem sehr expressiven Typ- und Modulsystem. Durch statische Typisierung gewinnt Alice gegenüber dem dynamisch getypten Mozart eine wichtige neue Qualität. Da ein Teil der Mozart-Funktionalität bisher nie statisch typisiert realisiert wurde, sind hier neue und interessante Forschungsaufgaben zu lösen. Statische Typisierung ist ein attraktives Designprinzip moderner Programmiersprachen, von dem sowohl die Anwender als auch die Entwickler der Sprache profitieren.

Es liegt bereits ein Design von Alice vor, der einige wesentliche Elemente der Mozart-Funktionalität abdeckt. Auf der Basis von Mozart wurde ein prototypische Implementierung dieser Version von Alice entwickelt, die als Plattform für Fallstudien dienen kann.

Das beantragte Projekt soll das Design von Alice (als Sprache) zu Ende führen. Dabei sind einige komplexere Fragen zu klären, für die Fallstudien durchgeführt werden müssen. Weiter soll das Projekt Alice (als Programmiersystem) auf zweifache Art realisieren. Die erste Realisierung baut auf der virtuellen Maschine von Mozart auf und kann bis Ende 2002 fertiggestellt werden. Für die zweite Realisierung soll eine neue virtuelle Maschine entworfen werden, die die überalterte Mozart-Maschine ersetzt. Analog zum Redesign der Sprache sollen hier die gesammelte Mozart-Erfahrung in einen neuen Maschinen-Design umgesetzt werden, der aus wiederverwendbaren Teilen besteht und einen Beitrag zur Forschung über die Implementierung von Programmiersystemen leistet.

3.5.1 AP1: Sprache, Typisierung, Fallstudien

Alice ist als konservative Erweiterung von SML konzipiert. Die Erweiterungen betreffen insbesondere Futures, Persistenz, Verteilung und Constraints. Diese Konzepte wurden in Mozart erfolgreich realisiert und sollen nun in verbesserter Form im Rahmen einer statisch getypten, funktionalen Sprache verfügbar gemacht werden. Da Oz eine dynamisch getypte, relationale Sprache ist, ergeben sich erhebliche Unterschiede bei der programmiersprachlichen Modellierung dieser Konzepte.

In diesem Arbeitspaket geht es um die Definition der Programmiersprache Alice, nicht um ihre Realisierung. Die Programmiersprache wird dabei als eine Formalismus gesehen, mit dem komplexe Abstraktionen modelliert werden können. Um zu verstehen, ob die Ausdrucksstärke der Sprache hinreichend ist, ist die Durchführung von Fallstudien für die neuen Anwendungsbereiche unerlässlich.

Aus programmiersprachlicher Sicht sind die notwendigen Erweiterungen des Typsystems besonders komplex. Typsysteme sind komplexe logische Spezifikationssprachen. Die bisher wichtigste Neuerung bei Alice ist die Einführung von Laufzeittypen (Pakete, dynamische Typprüfung). Damit scheint Alice in der Lage zu sein, die Funktionalität für Persistenz und Verteilung hinreichend flexibel zu modellieren.

Das Benutzungsmodell für SML ist klassisch und sieht vor, dass ein Programm als geschlossene Einheit vorliegt. Im Gegensatz dazu realisiert Alice (so wie Mozart) ein offenes Programmiermodell. Dabei können dynamisch Objekte hinzugenommen, die durch Berechnung in anderen Prozessen entstanden sind. Diese Objekte können Prozeduren enthalten, also dynamisch gebundene parametrisierte Programme. Die Integration dieser dynamisch erworbenen Programmteile gelingt mit den bereits erwähnten Laufzeittypen.

Hier sind einige wichtige Fragestellungen, die geklärt werden sollen:

1. Alice führt Typprüfungen zu drei verschiedenen Zeitpunkten durch: Beim Übersetzen von Komponenten, beim Laden von Komponenten, und beim Auspacken von Modulen. Der genaue Zusammenhang zwischen diesen Typprüfungen soll geklärt werden. Existierende SML-Systeme führen nur zum Übersetzungszeitpunkt Typprüfungen durch.
2. Eine Alice-Anwendung besteht aus Komponenten, die bedarfsgesteuert geladen werden (im Gegensatz zu existierenden SML-Systemen). Beim Laden einer Komponente muss geprüft werden, ob diese die erwartete Signatur erfüllt. Für diese Typprüfung existieren verschiedene Möglichkeiten. Für Alice soll eine möglichst flexible Lösung gefunden werden. Bisher fehlt eine Theorie, mit der diese Fragen geklärt werden können.
3. Sicherer Export. Objekte, deren Abschluss Objekte enthält, die zum Zustand der virtuellen Maschine oder des Betriebssystems gehören, können nicht exportiert werden. Eine Möglichkeit besteht darin, bei einem Exportversuch eine Ausnahme zu werfen. Besser wäre es, wenn man diese Laufzeitfehler durch statische Typprüfung ausschließen könnte. Wahrscheinlich kann dies durch ein monadisches Typsystem (wie in Haskell) geleistet werden, dies wäre aber ein radikaler Bruch mit SML. Es soll geprüft werden, ob es Möglichkeiten gibt, entsprechende statische Prüfungen mit einer konservativen Erweiterung von SML zu realisieren. Vorstellbar sind die Verwendung von Effektsystemen oder Typattributen.
4. Alice ist so wie SML als funktionale Programmiersprache konzipiert. Viele Mozart-Bibliotheken sind jedoch objektorientiert modelliert. Wir wollen verstehen, wie sie funktional modelliert werden können, und was die Trade-offs bei diesem Wechsel sind. Ein besonders interessanter Fall ist die Grafik-Bibliothek. Einige objektorientierte Konzepte lassen sich in funktionalen Sprachen mit Typklassen (wie in Haskell) modellieren. Falls damit signifikante Vorteile erzielt werden können, soll Alice (und damit SML) um Typklassen erweitert werden. Dazu müssen jedoch einige offene Fragen bezüglich Design und Semantik geklärt werden.
5. Es ist noch offen, wie Baumconstraints in einer statisch getypten Sprache modelliert werden können. Mithilfe von Fallstudien soll die Vorteile und Nachteile verschiedener Modellierungsmöglichkeiten geklärt werden. Mit einer Erweiterung des Typsystems um polymorphe Records scheint eine flexible getypte Modellierung möglich zu sein.
6. Um Alice an einer komplexen Aufgabenstellung zu erproben, die Verteilungs- und Constraintfunktionalität benötigt, soll ein verteilte Suchmaschine entwickelt werden. Verteilte Suchmaschinen lassen sich mit der Mozart-Funktionalität relativ einfach realisieren. Mo-

zart ist zur Zeit das einzige Programmiersystem, dass eine anwendungstaugliche verteilte Suchmaschine anbietet.

3.5.2 AP2: Übersetzer

Die Realisierung von Alice ist eine hochkomplexe Angelegenheit, die aus Informatiksicht beträchtlichen Forschungspotential hat. Wir streben eine modulare Realisierung an, die aus gut verstandenen und wiederverwendbaren Teilen besteht. Die grundlegende Systemteile sind der Übersetzer und die virtuelle Maschine. Der Übersetzer erzeugt Code für die virtuelle Maschine. Die virtuelle Maschine realisiert eine ungetypte Sprache. Kernaufgaben des Übersetzers sind also die statische Typüberprüfung, die Übersetzung hochsprachlicher Konstrukte wie Module und Typen in primitivere Konstrukte, sowie die eigentliche Code-Erzeugung.

Dynamische Typen. Alice erweitert SML um dynamische Typprüfungen beim Übergang von Paketen zu Modulen. Dafür müssen Typen und Signaturen dynamisch dargestellt werden. Außerdem muss die komplexe Matching-Operation für Signaturen realisiert werden. Da existierende SML-Systeme Typen zur Übersetzungszeit vollständig eliminieren (da sie dort zur Laufzeit nicht mehr benötigt werden), befinden wir uns hier auf unerforschtem Gebiet. Da Alice zudem ein höherstufiges Modulsystem verwendet, sind die Operationen für den Aufbau und Abgleich von Typen und Signaturen zum Teil sehr komplex. Die Realisierung von dynamischen Typen im Prototyp ist ad hoc, ineffizient und unvollständig. Es soll ein kompositionales Übersetzungsschema für Typen und Signaturen entwickelt werden, mit dem dynamische Typen effizient realisiert werden können.

Typüberprüfung von Zwischendarstellungen. Der Übersetzer verwendet mehrere Zwischendarstellungen, die zum überwiegenden Teil Typinformation tragen. Im Prinzip besteht die Möglichkeit, die Typkonsistenz der Zwischendarstellungen automatisch zu prüfen. Eine solche Prüfung könnte einige Korrektheitseigenschaften des Übersetzers automatisch verifizieren. Es ist zu klären, ob getypte Zwischensprachen im Alice-Übersetzer mit vertretbaren Aufwand realisierbar sind.

3.5.3 AP3: Virtuelle Maschine

Die virtuelle Maschine führt eine ungetypte Sprache aus, die über reiche Datenstrukturen verfügt. Möglichst viele Teile der virtuellen Maschine sollen sprachunabhängig realisiert sein. Hier sind einige Designideen:

1. Die virtuelle Maschine realisiert eine plattformunabhängige Schnittstelle für die Ausführung von Programmkomponenten. Programmkomponenten werden bedarfsgesteuert geladen.
2. Ein sprachunabhängiger *abstrakter Speicher* nimmt alle Objekte (auch Code und Threads) auf, mit denen die virtuelle Maschine rechnet. Der Speicher ist mit einer automatischen Bereinigung realisiert. Die sprachlichen Daten- und Ablaufstrukturen werden auf die Datenstrukturen des Speichers zurückgeführt.
3. Die sogenannte *Transfersprache* definiert eine Plattform-unabhängige externe Darstellung für die Objekte, die im abstrakten Speicher dargestellt werden können. Für die Darstellung von Code beinhaltet die Transfersprache einen abstrakten Instruktionssatz.
4. Der *Serialisierer* konvertiert zwischen der internen und externen Darstellung von Objekten. Die interne Darstellung erfolgt im abstrakten Speicher. Die externe Darstellung erfolgt in der Transfersprache. Externe Darstellungen können zwischen Prozessen kommuniziert werden (Verteilung) und in Dateien abgelegt werden (Persistenz).
5. Der *Interpreter* führt Code gemäß eines internen Instruktionssatzes aus. Mithilfe eines *Laufzeitübersetzers* übersetzt die Maschine abstrakten Code in internen. Eine Maschine kann mit mehreren Codes und Interpretern arbeiten. Die Trennung von abstraktem und internen Instruktionssatz entkoppelt die programmiersprachlichen Aspekte (abstrakter Instruktionssatz) von architekturspezifischen Aspekten (interner Instruktionssatz). Neben einer generellen Reduktion an Komplexität erweitert dieses Vorgehen die Einsatzbreite der virtuellen Maschine beträchtlich. So sind interne Instruktionssätze möglich, die bezüglich Platz (schlanker Instruktionssatz für mobile Geräte wie PDAs), Effizienz (reicher oder sogar nativer Instruktionssatz für Desktop Computer), oder Ablaufinformation (Unterstützung bei der Programmentwicklung) optimiert sind.
6. Der *Scheduler* realisiert die globale Steuerung der Maschine. Er entscheidet, welcher Thread ausgeführt wird und ruft den Laufzeitübersetzer und den Interpreter auf.
7. Der *Port* ist für die Kommunikation mit anderen laufenden Maschinen (Prozesse) zuständig. Er empfängt und sendet Objekte in externer Darstellung.

Während andere Systeme (Java, Erlang) getrennte externe Darstellungen für Code und Datenstrukturen verwenden, arbeitet Alice wie Mozart nur mit einer uniformen Darstellung (die natürliche Lösung für Sprachen mit emanzipierten Prozeduren). Die uniforme Darstellung ist flexibler und vermeidet die mit getrennten Darstellungen verbundenen Probleme.

3.5.4 AP4: Laufzeitsystem

Einige Teile der virtuellen Maschinen sollen auf hoher Ebene in Alice programmiert werden. Die Gesamtheit dieser Teile wird als Laufzeitsystem bezeichnet. Dazu gehören der Komponentenmanager, die dynamische Typprüfung, der Laufzeitübersetzer und einige grundlegende Bibliotheken.

Der Komponentenmanager ist für die Beschaffung und Verwaltung der Programmkomponenten zuständig. Damit der Komponentenmanager in Alice geschrieben werden kann, müssen typisierbare reflektive Schnittstellen entworfen werden. Diese Problematik ist bisher ungeklärt.

Wenn die Maschine startet, muss das Laufzeitsystem schrittweise geladen und installiert werden. Hier sind eine Reihe von zyklischen Abhängigkeiten aufzulösen. Die Auflösung geschieht dadurch, dass man mit einem sehr primitiven Laufzeitsystem beginnt, dass schrittweise dass vollständige Laufzeitsystem aufbaut. Einige Komponenten müssen dabei in mehreren Varianten vorliegen, wobei primitivere Varianten schrittweise durch Varianten mit mehr Funktionalität ersetzt werden.

3.5.5 AP5: Bibliotheken

Die Bibliotheken stellen im Rahmen der Sprache Funktionalität zur Verfügung, die für viele Anwendungen wiederverwendet werden kann. Von besonderen Interesse sind bei Alice Bibliotheken für grafische Benutzerschnittstellen und Constraints. Bibliotheken werden gemischt realisiert, teilweise in Alice und teilweise auf tieferer Ebene in C++. Die Schnittstelle einer Bibliothek muss die angebotene Funktionalität mit Alice modellieren. Die Modellierung ist sowohl für Grafik als auch Constraints interessant, da diese Funktionalitäten heute typischerweise in dynamisch getypten oder objektorientierten Sprachen angeboten werden. Die Grafik-Funktionalität kann aus existierenden Systemen importiert werden und erfordert daher nur wenig Entwicklungsaufwand. Die Constraintfunktionalität wollen wir mithilfe der für Mozart geschriebenen Bibliotheken realisieren.

3.5.6 AP6: Werkzeuge

Mozart stellt eine Reihe von neuartigen Programmierwerkzeugen zur Verfügung, die für Anwender eine wichtige Rolle spielen. Dazu gehören insbesondere der Explorer (Constraintpro-

grammierung) und der Inspector. Der Inspector ist ein nebenläufiger Browser, mit dem alle Laufzeitobjekte dargestellt werden können.

Für Alice soll der Inspector so erweitert werden, dass auch Module und Typen dargestellt werden können. Der erweiterte Inspector soll in Alice programmiert werden. Damit bekommen wir eine interessante Fallstudie, mit der wir die Modellierung der Grafikbibliothek testen können. Da der Inspector in Mozart größtenteils mit objektorientierten Sprachmitteln realisiert ist, die es so in Alice nicht gibt, liefert uns die Übertragung nach Alice eine interessante Fallstudie für die Trade-offs zwischen funktionaler und objektorientierter Programmierung.

Zwei weitere wichtige Werkzeuge sind der Interpreter und der Binder. Der Interpreter ist ein interaktives Werkzeug zum experimentellen Programmieren. Mit dem Binder können mehrere Komponenten in eine Komponente kombiniert werden, die schneller geladen werden kann als die Einzelkomponenten (kleine Komponenten bei der Entwicklung, große Komponenten bei der fertigen Anwendung). Der Binder ist eine interessante Fallstudie für statisch getypte Programmierung, da er die Typkonsistenz der Einzelkomponenten prüfen muss und dafür reflektive Schnittstellen erforderlich sind.

3.5.7 Zeitplan

Die Arbeitspakete sollen alle ab Projektanfang bearbeitet werden. Die meisten Arbeitspakete werden bereits in der laufenden Förderperiode vorbereitet. Bis Ende 2002 soll die Mozart-basierte Realisierung von Alice für Anwender verfügbar sein.

3.6 Stellung innerhalb des Programms des Sonderforschungsbereichs

NEP erfüllt im SFB eine wichtige Infrastrukturfunktion, indem es das Paradigma der nebenläufigen Constraintprogrammierung theoretisch und praktisch einbringt. In der laufenden Förderperiode besteht eine enge Kooperation mit den Projekten B1, C2, C3 und C4. Diese benutzen das von NEP weiterentwickelte Programmiersystem Mozart, vor allem wegen seiner innovativen Funktionalität in den Bereichen Constraints, Nebenläufigkeit, Verteilung und Persistenz. Entsprechend den Anforderungen der kooperierenden Teilprojekte wurde Mozart um neue Funktionalität erweitert. Für C2, C3 und C4 wurde eine leistungsfähige Bibliothek für Mengenconstraints entwickelt. Für B1 wurden first-class Constraints hinzugefügt. Die bestehenden Kooperationen sollen mit den Projekten MI 2 (Nachfolger von C4), MI 3 (Nachfolger von C2) und MI 4 (Nachfolger von B1) fortgesetzt werden. Mit dem neu beantragten Projekt

MI 5 ist eine enge Kooperation geplant. Da Logical Frameworks traditionell in ML implementiert werden, ist MI 5 der ideale Kandidat für die erste echte Anwendung von Alice.

Literatur

Abadi, M., Cardelli, L., Pierce, B. C. & Plotkin, G. D. (1989). Dynamic typing in a statically-typed language. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages, POPL'89* (S. 213–227). Austin, Texas.

Abadi, M., Cardelli, L., Pierce, B. C. & Rémy, D. (1995). Dynamic typing in polymorphic languages. *Journal of Functional Programming, 5(1)*, 111–130.

Alice Team. (2001). *The Alice System.* http://www.ps.un-sb.de/alice/.

Aït-Kaci, H. (1991). *Warren's abstract machine: A tutorial reconstruction.* Cambridge, MA, USA: The MIT Press.

Bekkers, Y., Canet, B., Ridoux, O. & Ungaro, L. (1986). MALI: A memory with a real-time garbage collector for implementing logic programming languages. In *Proceedings of theInternational Symposium on Logic Programming* (S. 258–265). The Computer Society Press.

Boehm, H.-J. & Weiser, M. (1988). Garbage collection in an uncooperative environment. *Software Practice and Experience, 18*(9), 807–820.

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

Crary, K., Weirich, S. & Morisett, G. (1998). Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming, ICFP'98.* Baltimore, Maryland.

Dubois, C., Rouaix, F. & Weis, P. (1995). Extensional polymorphism. In *Proceedings of the 22th ACM Conference on Principles of Programming Languages, POPL'95.*

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)

Flatt, M. & Felleisen, M. (1998). Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)* (S. 236–248). Montreal, Canada.

Garrigue, J. (1998). Programming with polymorphic variants. In *Proceedings of ML Workshop.* Baltimore, Maryland.

Garrigue, J. (2000). Code reuse through polymorphic variants. In *Proceedings of Workshop on Foundations of Software Engineering.* Sasaguri, Japan.

Gaster, B. R. & Jones, M. P. (1996). *A polymorphic type system for extensible records and variants* (Tech. Rep.). University of Nottingham.

Gifford, D. K. & Lucassen, J. M. (1986). Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming* (S. 28–38). Cambridge, Massachusetts: ACM Press.

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.

Harper, R. & Lillibridge, M. (1994). A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'94* (S. 123–137). Portland, OR.

Harper, R., Mitchell, J. C. & Moggi, E. (1990). Higher-order modules and the phase distinction. In *Proc. of 17th ACM Symposium on Principles of Programming Languages, POPL'90* (S. 341–354).

Henderson, F., Conway, T., Somogyi, Z., Jefferey, D., Schachte, P., Taylor, S. & Speirs, C. (2000). *The Mercury language reference manual*. http://www.cs.mu.oz.au/research/mercury/information/documentation.html.

Hinze, R. & Peyton Jones, S. (2000). Derivable type classes. In G. Hutton (Hrsg.), *Proceedings of the Forth Haskell Workshop*. Montreal, Canada.

Jeffery, D., Dowd, T. & Somogyi, Z. (1999). MCORBA: A CORBA binding for Mercury. *Lecture Notes in Computer Science, 1551*, 211–227.

Jones, M. P. (1993). A system of constructor classes: Overloading and hmplicit higher-order polymorphism. In *Functional Programming and Computer Architecture, FPCA'93* (S. 52–61). ACM Press.

Jones, M. P. (1994). *Qualified types: Theory and practice*. Dissertation, Oxford University. (Also available as Programming Research Group technical report 106)

Jones, M. P. (1997). First-class polymorphism with type inference. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages, POPL'97*. Paris, France.

Jones, M. P. (2000). Type classes with functional dependencies. In G. Smolka (Hrsg.), *Proceedings of the 9th European Symposium on Programming, ESOP 2000* (Bd. 1782). Berlin, Germany: Springer-Verlag.

Jones, M. P. & Peyton Jones, S. (1999). Lightweight extensible records for haskell. In E. Meijer (Hrsg.), *Proceedings of the 1999 Haskell Workshop*. Paris, France: University of Utrecht.

Jones, M. P., Reid, A. & Others. (2001). *The Hugs 98 user manual*. http://www.haskell.org/hugs/.

Jones, R. & Lins, R. (1996). *Garbage collection algorithms for automatic dynamic memory management*. New York, NY, USA: John Wiley & Sons, Inc.

Kaes, S. (1988). Parameteric overloading in polymorphic programming languages. In *European Symposium on Programming Languages and Systems, ESOP'88* (Bd. 300, S. 131–141). Springer-Verlag.

Läufer, K. (1996). Type classes with existential types. *Journal of Functional Programming, 6*(3), 485–517.

Leroy, X. (1990). *The* ZINC *experiment: an economical implementation of the ML language* (Technical report Nr. 117). INRIA.

Leroy, X. (1995). Applicative functors and fully transparent higher-order modules. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.* (S. 142–153). New York, NY.

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

man/.

Leroy, X. & Mauny, M. (1993). Dynamics in ML. 431–463.

Lewis, J., Shields, M., Meijer, E. & Launchbury, J. (2000). Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2000* (S. 108–118). Boston, Massachusetts.

Lillibridge, M. (1997). *Translucent sums: A foundation for higher-order module systems.* Dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Lindholm, T. & Yellin, F. (1997). *The Java virtual machine specification.* Reading, MA, USA: Addison-Wesley.

Lucent. (2000). *Standard ML of New Jersey user's guide.* http://cm.bell-labs.com/cm/cs/what/smlnj/doc/.

MacQueen, D. B. (1986). Using dependent types to express modular structure. In *Proc. of 1986 ACM Symposium on Principles of Programming Languages* (S. 277–286). St. Petersburg.

MacQueen, D. B. & Tofte, M. (1994). A semantics for higher-order functors. In D. Sannella (Hrsg.), *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming* (Bd. 788, S. 409–423). Edinburgh, U.K.: Springer.

Marlow, S., Peyton Jones, S. & Others. (1999). *The Glasgow Haskell Compiler.* http://www.haskell.org/ghc/.

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

Mehl, M., Scheidhauer, R. & Schulte, C. (1995). An abstract machine for Oz. In M. Hermenegildo & S. D. Swierstra (Hrsg.), *Programming Languages, Implementations, Logics and Programs, Seventh International Symposium, PLILP'95* (Bd. 982, S. 151–168). Utrecht, The Netherlands: Springer-Verlag.

Microsoft. (2000). *Microsoft .NET: Realizing the next generation internet* (Tech. Rep.). Microsoft. (Available from http://www.microsoft.com/business/vision/netwhitepaper.asp)

Müller, T. (2000). Practical investigation of constraints with graph views. In R. Dechter (Hrsg.), *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming* (Bd. 1894, S. 320–336). Singapore: Springer-Verlag.

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/`)

Nielson, F. & Nielson, H. R. (1999). Type and Effect Systems. In E. R. Olderog & B. Steffen (Hrsg.), *Correct system design* (S. 114–136). Springer-Verlag.

Nipkow, T. & Prehofer, C. (1995). Type reconstruction for type classes. 201–224.

Ogawa, H., Shimura, K., Matsuoka, S., Maruyama, F., Sohda, Y. & Kimura, Y. (2000). OpenJIT: An open-ended, reflective JIT compiler framework for java. In E. Bertino (Hrsg.), *Ecoop 2000 - object-oriented programming, 14th european conference* (Bd. 1850, S. 362-387). Sophia Antipolis and Cannes, France: Springer-Verlag.

Ohori, A. (1995). A polymorphic record calculus and its compilation. In *ACM Transactions on Programmoing Languages and Systems* (Bd. 17, S. 844-895).

Okasaki, C. (2000). An overview of edison. In G. Hutton (Hrsg.), *Proceedings of the Forth Haskell Workshop.* Montreal, Canada.

Peyton Jones, S. (1996). Bulk types with class. In *Proceedings of the 1996 Workshop on Functional Programming.* Ullapool, Scotland.

Peyton Jones, S., Hughes, J. et al.. (1998). *Haskell 98: A non-strict, purely functional language* (Technical Report). http://www.haskell.org/onlinereport/.

Peyton Jones, S., Jones, M. P. & Meijer, E. (1997). Type classes: an exploration of the design space. In *Proceedings of the Haskell Workshop* (Bd. 788).

Peyton Jones, S. & Wadler, P. (1993). Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages, POPL'93.* Charlotte, North Carolina.

Pottier, F. (1998). A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)* (S. 228–238). Baltimore, USA.

Pottier, F. (2000). A 3-part type inference engine. In G. Smolka (Hrsg.), *Proceedings of the 2000 European Symposium on Programming (ESOP'00)* (Bd. 1782, S. 320–335). Berlin, Germany: Springer Verlag.

Rémy, D. (1993). Type inference for records in a natural extension of ML. In C. A. Gunter & J. C. Mitchell (Hrsg.), *Theoretical aspects of object-oriented programming. types, semantics and language design.* MIT Press.

Rémy, D. & Vouillon, J. (1998). Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems, 4*(1), 27–50. (A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997)

Romanenko, S., Russo, C. & Sestoft, P. (2000a). *Moscow ML language overview.* ftp://ftp.dina.kvl.dk/pub/mosml/doc/mosmlref.pdf.

Romanenko, S., Russo, C. & Sestoft, P. (2000b). *Moscow ML owner's manual.* ftp://ftp.dina.kvl.dk/pub/mosml/doc/manual.pdf.

Russo, C. V. (1998). *Types for modules.* Dissertation, University of Edinborgh.

Russo, C. V. (1999). Non-dependent types for Standard ML modules. In *1999 International Conference on Principles and Practice of Declarative Programming, PPDP.* Paris, France.

Russo, C. V. (2000). First-class structures for Standard ML. In G. Smolka (Hrsg.), *Proceedings of the 9th European Symposium on Programming, ESOP 2000* (Bd. 1782). Berlin, Germany: Springer-Verlag.

Sage, M. (2000). FranTk - a declarative GUI language for Haskell. In *Proceedings of Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000* (S. 106–118). Montreal, Canada: ACM Press.

Scheidhauer, R. (1998). *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz.* Doctoral dissertation, Universität des Saarlandes, Im Stadtwald, 66041 Saarbrücken, Germany. (In German.)

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.

Sewell, P. (2001). Modules, abstract types, and distributed versioning. In *Proceedings of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages, POPL'01*. London, UK.

Sulzmann, M. (2000). *A general framework for Hindley/Milner type systems with constraints.* Dissertation, Yale University, Department of Computer Science.

Sulzmann, M., Odersky, M. & Wehr, M. (1999). Type inference with constrained types. *Theory and Practice of Object Systems, 5(1).*

Talpin, J.-P. & Jouvelot, P. (1994). The type and effect discipline. *Information and Computation*, *111*(2), 245–296.

Tofte, M. & Birkedal, L. (1998). A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, *20*(3), 1–44.

Van Roy, P., Haridi, S., Brand, P., Smolka, G., Mehl, M. & Scheidhauer, R. (1997). Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, *19*(5), 804–851.

Wadler, P. (1992). The essence of functional programming. In *19th ACM Symposium on Principles of Programming Languages, POPL'92*. Albuquerque, New Mexico.

Wadler, P. (1998). The marriage of effects and monads. In *Proceedings of the International Conference on Functional Programming, ICFP'98.* Baltimore, Maryland.

Wadler, P. & Blott, S. (1989). How to make *ad-hoc* polymorphism less *ad hoc*. In *16th ACM Symposium on Principles of Programming Languages* (S. 60–76).

Warren, D. H. D. (1983). *An abstract Prolog instruction set* (Technical Note Nr. 309). Menlo Park, CA, USA: SRI International, Artificial Intelligence Center.

Wilson, P. R. (1992). Uniprocessor garbage collection techniques. In Y. Bekkers & J. Cohen (Hrsg.), *Memory Management: International Workshop IWMM 92* (Bd. 637, S. 1–42). St. Malo, France: Springer-Verlag. (Revised version will appear in ACM Computing Surveys)

Würtz, J. (1998). *Lösen kombinatorischer probleme mit constraintprogrammierung in Oz.* Dissertation, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany.

3.7 Ergänzungsausstattung für das Teilprojekt MI 6

PK: Personalbedarf und -kosten (Begründung vgl. 3.7.1)
SV: Sächliche Verwaltungsausgaben (Begründung vgl. 3.7.2)
I: Investitionen (Geräte über DM 20.000 brutto; Begründung vgl. 3.7.3)

	Bewilligung 2001			**2002**			**2003**			**2004**		
PK	Verg.-Gr.	Anz.	Betrag DM	Verg.-Gr.	Anz.	Betrag DM	Verg.-Gr.	Anz.	Betrag DM	Verg.-Gr.	Anz.	Betrag DM
	BAT IIa	2	201 600	BAT IIa	2	213 600	BAT IIa	2	213 600	BAT IIa	2	213 600
	SHK	2	38 400	SHK	2	40 800	SHK	2	40 800	SHK	2	40 800
	zus.:	4	240 000	zus.:	4	254 400	zus.:	4	254 400	zus.:	4	254 400

SV	Kosten-kategorie oder Kennziff.	Betrag DM	Kosten-kategorie oder Kennziff.	Betrag DM	Kosten-kategorie oder Kennziff.	Betrag DM
		0		0		0
	zus.	0	zus.	0	zus.	0

I	Investitionsmittel insges.	Investitionsmittel insges.	Investitionsmittel insges.
	0	0	0

3.7.1 Begründung des Personalbedarfs

	Name, akad. Grad, Dienststellung	engeres Fach des Mitarbeiters	Institut der Hochschule oder der außeruniv. Einrichtung	Mitarbeit im Teilprojekt in Std./Woche (beratend: B)	auf dieser Stelle im SFB tätig seit	beantragte Einstufung in BAT
Grundausstattung						
3.7.1.1 wissenschaftl. Mitarbeiter (einschl. Hilfskräfte)	1. Smolka, Gert, Prof. Dr. 2. Schulte, Christian, Dr.. 3. Brunklaus, Thorsten, Dipl.-Inform.	Informatik Informatik Informatik	FR Informatik FR Informatik FR Informatik	8 12 16	01/1996 04/1999 03/2000	– – –
3.7.1.2 nichtwissenschaftl. Mitarbeiter	4. Hussung, Bärbel, Verwaltungsangestellte	–	FR Informatik	4	01/1996	–
Ergänzungsausstattung						
3.7.1.3 wissenschaftliche Mitarbeiter (einschl. Hilfskräfte)	5. Rossberg, Andreas, Dipl.-Inform. 6. N.N. 7. N.N. (SHK) 8. N.N. (SHK)	Informatik Informatik Informatik Informatik	FR Informatik FR Informatik	38.5 38.5 19 19	03/2000 07/1997	BAT IIa BAT IIa SHK SHK
3.7.1.4 nichtwissenschaftl. Mitarbeiter						

(Stellen, für die Mittel *neu* beantragt werden, sind mit X gekennzeichnet.)

Aufgabenbeschreibung von Mitarbeitern der Grundausstattung

1 Projektleitung.

2 Herr Schulte ist einer der Hauptentwickler des Mozart-Systems. Er wird alle Arbeitspakete unterstützen, insbesondere bei Fragen, die Constraints und die Realisierung der abstrakten Maschine betreffen.

3 Herr Brunklaus soll hauptsächlich zu den Arbeitspaketen 3-6 beitragen.

4 Sekretariatsaufgaben.

Aufgabenbeschreibung von Mitarbeitern der Ergänzungsausstattung

5 Herr Rossberg soll hauptsächlich zu den Arbeitspaketen 1, 2 und 4 beitragen.

6 Dieser wissenschaftliche Mitarbeiter ist für die Arbeitspakete 3-6 zuständig.

7+8 Die studentischen Mitarbeiter unterstützen die wissenschaftlichen Mitarbeiter bei den umfangreichen Implementierungsarbeiten und bei den Fallstudien.

3.7.2 Aufgliederung und Begründung der Sächlichen Verwaltungsausgaben (nach Haushaltsjahren)

	2002	2003	2004
Für Sächliche Verwaltungsausgaben stehen als *Grundausstattung* voraussichtlich zur Verfügung:	2 000	2 000	2 000
Für Sächliche Verwaltungsausgaben werden als *Ergänzungsausstattung* beantragt (entspricht den Gesamtsummen „Sächliche Verwaltungsausgaben“ in Übersicht 3.7)	vgl. Z	vgl. Z	vgl. Z

(Alle Angaben in DM.)

Begründung zur *Ergänzungsausstattung* der Sächlichen Verwaltungsausgaben

Siehe Teilprojekt Z.

3.7.3 Investitionen (Geräte über DM 20.000,— brutto und Fahrzeuge)

Entfällt.