Definition einer spezialisierten Metaobjektklasse für parallele Instanzenverwaltung

Lothar Hotz

Universität Hamburg
Fachbereich Informatik
Bodenstedtstr. 16
2000 Hamburg 50
e-mail: lothar@ki5.informatik.uni-hamburg.de

Kurzfassung

Dieses Papier beschreibt, wie das CLOS Metaobjekt Protokoll genutzt werden kann, um in einfacher Weise parallele Maschinen zu programmieren. Über dieses Protokoll wird die Implementation der objektorientierten auf Common-Lisp basierenden Programmiersprache CLOS spezifiziert. Dadurch werden Erweiterungen und Modifikationen von CLOS ermöglicht. Die objektorientierte Definition des Metaobjekt Protokolls unterstützt dabei die Spezifikation neuer Konstrukte. Sie werden als spezialisierte Metaobjektklassen in das Protokoll eingehängt. Dieser Ansatz wird hier verwendet, um Klassen definieren zu können, deren Instanzen in parallelarbeitenden Prozessoren gespeichert werden.

1 Einleitung

In diesem Papier wird untersucht, wie die objektorientierte Programmiersprache CLOS [Keene, 1989] erweitern kann, um sie für die Programmierung der Connection Machine (CM) einzusetzen. Die Idee dabei ist, einem CLOS-Programmierer Hilfsmittel bereitzustellen, die es ihm erlauben, die parallelen Konzepte der CM auszunutzen, ohne sie explizit bei der Implementierung berücksichtigen zu müssen. Dazu sollen die Möglichkeiten des Metaobjekt Protokolls (MOP) angewandt werden.

In [Kiczales et al., 1991] wird für eine Untermenge von CLOS (Closette) das Metaobjekt Protokoll spezifiziert (vgl. auch [Kiczales und Bobrow, 1988], [Bobrow und Kiczales, 1988]). In ihm werden die Konzepte von CLOS (Klasse, Slot, generische Funktion, Methode, Spezialisierer, Methodenkombination) explizit gemacht, um nicht nur *eine* objektorientierte Programmiersprache definieren zu können, sondern ein Bereich von Sprachen zu beschreiben, der unterschiedliche Ausprägungen der Konzepte abdeckt. So werden in [Kiczales et al., 1991] u.a. Erweiterungen von CLOS durch Facetten, Klassen, die ihre Instanzen in Hashtables verwalten, Berechnung unterschiedlicher Klassenpräzedenzlisten, Einfügen von protokollierbaren Methoden (Tracing), verdeckte (encapsulated) Methoden und verschiedene Arten der Methodenkombination beschrieben. In [Paepcke, 1990] wird eine Erweiterung von CLOS mit Hilfe des MOP in Hinsicht auf persistente Objekte, die in Datenbanken gespeichert werden, verfolgt. All diese Erweiterungen werden durch das MOP ermöglicht, indem die Spezifikation der internen Strukturen und Abläufe (d.h. die CLOS-Implementation) über Metaobjektklassen zur Verfügung steht. Je nach Erweiterung werden neue spezialisierte Metaobjektklassen unter die gegebenen eingefügt. Für die Einführung einer weiteren Berechnungsart der Präzedenzlisten wird z.B.

eine Metaobjektklasse gebildet, die die CLOS-Implementation dort verändert, wo diese Liste berechnet wird (in *compute-class-precedence-list).*

Die vorangegangenen Beispiele fortsetzend, werden wir nach einer kurzen Beschreibung des MOP, die Definition und Implementation der Metaobjektklasse *parallel-slot-class* beschreiben. Sie erlaubt es parallele Slots zu definieren. Werte von Instanzen eines solchen Slots werden in unterschiedlichen Prozessoren gespeichert und so parallel verarbeitbar. Da Instanzen lediglich eine bestimmte Ansammlung von Slotwerten beschreiben, kann man von paralleler Instanzenverwaltung sprechen. Dazu wurde *Lisp, ein Lisp-Dialekt für die Programmierung der CM [*Lisp, 1988] um Closette erweitert (CLOS selbst ist nicht in *Lisp enthalten). Das hierfür in [Kiczales et al., 1991] spezifizierte MOP wurde zur Implementation der Metaobjektklasse herangezogen. Die verwendeten Konstrukte von *Lisp werden kurz vorgestellt.

Die Metaobjektklasse *parallel-slot-class* soll zu einer Menge von spezialisierten Klassen gehören, die unter dem Namen *CLOS zusammengefaßt werden. Sie dienen dazu, die Rechenleistung der CM auszunutzen, ohne in *Lisp direkt zu programmieren. Stattdessen gibt ein CLOS-Programmierer lediglich die gewünschten Metaobjektklassen an, und diese verwalten die internen parallelen Strukturen der CM. Alternativ dazu müßte man *Lisp-Programme schreiben und die parallelen Konzepte (z.B. die Verwendung paralleler Variablen) erlernen und direkt verwenden. Statt der tabellenartigen Struktur solcher Variablen, sollen bei der Modellierung eines Problems komplexere Strukturen, wie Folgen, Mengen, Relationen und Graphen, direkt verwendet werden können. Diese werden in *CLOS über Metaobjektklassen definiert (sie verwenden intern natürlich wieder parallele Variablen) und ermöglichen so dem Programmierer eine andere Sichtweise auf die Strukturen - er erhält durch *CLOS die Möglichkeit auf einer höheren Abstraktionsstufe zu programmieren.

Liegt etwa eine Folge von Elementen (z.B. ein String als Folge von Buchstaben) vor und sollen daraus bestimmte aufeinanderfolgende Elemente zu einem Aggregat zusammengefaßt werden (z.B. durch Leerstelle getrennte Buchstaben zu einem Wort), so kann dies durch eine Funktion geschehen, die die Folge und ein Prädikat (z.B. Erkennung der Leerstelle) verwendet, um Elemente zusammenzufassen. Die Folge wird mit einer Metaobjektklasse implementiert, für die eine solche Funktion vorhanden ist. Der *CLOS Programmierer benutzt die aggregierende Funktion. Auf einer tieferen Abstraktionsebene verwendet die Funktion die diesem Problem inhärente Parallelität. (Die Folgenelemente werden in eine Kette von Prozessor gespeichert. Durch Anwendung des Prädikats auf den vorhergehenden Prozessor wird erkannt, ob ein neues Aggregat vorliegt oder nicht. Dies kann für alle Elemente gleichzeitig geschehen.)

Andere Arbeiten, die Relationen und Graphen als Datenstrukturen liefern, werden z.B. in [Anthonisse, 1973], [Boley, 1980], [Boley, 1990], [Dewar et al. 1986], [Friedman, 1968] und [Rheinboldt und Basili, 1972] beschrieben. Diese Ansätze verwalten allerdings die Strukturen nicht parallel. Auf der anderen Seite gibt es objektorientierte Sprachen, die parallele Verarbeitung zulassen, dies aber durch neue Konstrukte realisieren (z.B. [Treleaven, 1988], [Annot et al., 1988]). In *CLOS sollen demgegenüber komplexe Strukturen eingeführt werden und die Operationen auf diesen intern parallelisiert werden, ohne daß man explizit parallel denken oder programmieren muß.

Um in diese Richtung einen Anfang zu machen, werden wir hier zeigen, wie grundsätzlich das MOP dazu verwendet werden kann, parallele Strukturen in CLOS einzuführen.

2 Kurze Beschreibung des Metaobjekt Protokolls

Im MOP werden für die grundlegenden Konzepte von CLOS Beschreibungsmittel eingeführt, die die Struktur und das Verhalten dieser Konzepte verdeutlichen. Sie werden Metaobjekte

Information	Repräsentation	einige generische Funktionen
Name	Objekt	class-name
direkte Unterklassen	Liste von Klassen-Metaobjekten	class-direct-subclass
direkte Oberklassen	Liste von Klassen-Metaobjekten	class-direct-superclass
Präzedenzliste	Liste von Klassen-Metaobjekten	class-precedence-list
direkte Slots	Liste von direkten Slotdefinitions-Metaobjekten	class-direct-slots
effektive Slots	Liste von effektiven Slotdefinitions-Metaobjekten	
Dokumentation	String	
Methoden mit dieser Klasse als Spezialisierer	Liste von Methoden-Metaobjekten	class-direct-methods make-instance allocate-instance initialize-instance :after compute-class-precedence-list compute-slots compute-effective-slot-definitions finalize-inheritance slot-value-using-class

Abbildung 1: Klassen-Metaobjekte beschreiben die Struktur der Instanzen und ihr Default-Verhalten

genannt, da sie Informationen über die Objekte von CLOS enthalten. Metaobjekte werden wiederum mit Hilfe der CLOS-Konzepte (Klassen, Methoden usw.), die sie beschreiben, implementiert, d.h. CLOS wird mit CLOS implementiert. Diese Eigenschaft wird *metazirkulär* genannt (vgl. auch [Paepcke, 1990], [Steele und Sussman, 1978]). Metaobjekte sind so Instanzen von Metaobjektklassen. Klassen-Metaobjekte beschreiben die Struktur von Klassen, generische Funktions-Metaobjekte beschreiben generische Funktionen und Methoden-Metaobjekte beschreiben Methoden. Einige vordefinierte Metaobjektklassen sind *standard-class, standard-generic-function, standard-method*; spezialisierte Metaobjektklassen können unter diese eingeordnet werden.[1]

Für die Metaobjektklassen werden generische Funktionen definiert, die ihr Verhalten und damit das Verhalten von CLOS beschreiben. In den Abbildungen 1 und 2 wird eine Übersicht für die Klassen und generischen Funktions-Metaobjekte, die in ihnen enthaltene Informationen und die wichtigsten generischen Funktionen, die sie betreffen, gegeben. Die Repräsentation verdeutlicht die Zusammenhänge der Klassen untereinander. Die genannten generischen Funktionen liefern die Ansatzpunkte für die Definition spezialisierter Metaobjektklassen. Die Abbildungen vermitteln damit einen Eindruck, welche Funktionalität mit Hilfe des MOP verändert werden kann.

Durch das Einhängen weiterer Methoden bzgl. spezialisierter Metaobjektklassen zu den gegebenen generischen Funktionen (vgl. Abbildungen 1 und 2) wird das Verhalten von CLOS modifiziert. *allocate-instance*, *initialize-instance :after* und *slot-value-using-class* werden wir für unser Beispiel neu definieren.

[1] Klassen-Metaobjektklassen werden auch Metaklassen genannt. Um jedoch die Unterscheidung zu den andern genannten Metaobjektklassen zu bekommen, sollte nach [Kiczales et al., 91, 1975] dieser Begriff nicht mehr verwendet werden.

Information	Repräsentation	einige generische Funktionen
Name	Funktionsname	generic-function-name
Zugeordnete Methoden	Liste von Methoden-Metaobjekten	generic-function-methods
lambda-Liste	Liste	generic-function-lambda-list
Methodenkombination	Methodenkombinations-Metaobjekt	
Dokumentation	String	
Deklarationen	Liste von Lisp Deklarationen	
		compute-applicable-methods
		compute-applicable-methods-using-class
		compute-discriminating-function
		compute-effective-method-function
		finalize-generic-function

Abbildung 2: Generische Funktions-Metaobjekte beschreiben den Auswahlmechanismus von Methoden

3 Eine spezialisierte Metaobjektklasse für die parallele Instanzenverwaltung

Wir wollen die Werte eines oder mehrerer Slots aller Instanzen einer Klasse parallel verarbeiten. Dazu betrachten wir kurz das hier verwendete Konzept der parallelen Variablen (PVARS, sprich "pivars") aus *Lisp. PVARS beschreiben eine Sammlung von Werten, die jeder in einem Prozessor der CM gespeichert werden. Eine PVAR muß vor ihrer Benutzung alloziert werden. Dies geschieht in *Lisp mit **defvar, allocate!!* oder bei Struktur-PVARS mit *make-...!!*, ähnlich wie bei *defstruct* in Common-Lisp. PVARS können Werte beliebiger unterschiedlicher Lisp-Typen beinhalten. Sind sie von einem einzigen Typ, dann können sie allerdings effizienter verarbeitet werden. Wir werden unten die spezialisierte Metaobjektklasse für typisierte PVARS definieren. Neben den gängigen Operationen, wie arithmetische oder logische Operationen auf zwei PVARS, können Prozessoren über Selektionsoperationen für die Berechnung ausgewählt werden. Weiterhin können einzelne Prozessoren über *pref* und einem Index, analog zu *aref* auf Arrays, angesprochen werden. Da wir im folgenden lediglich die verteilte Speicherung von Slotwerten in unterschiedlichen Prozessoren betrachten werden, wollen wir hier nicht auf weitere Operationen eingehen (vgl. [*Lisp, 1988]).

Durch Angabe einer Allokationsform für einen Slot, die einem PVAR-Typ entspricht, soll spezifiziert werden, daß die zu dem Slot gehörenden Werte aller Instanzen in einer PVAR gehalten werden. Nennen wir solche Slots *parallele Slots*. Mit:

```
(defclass many-instances ()
  ((state :allocation '(array (unsigned-byte 16) 50)))
  (:metaclass parallel-slot-class))
```

wird die Klasse *many-instances* definiert, deren Slot *state* eine parallele Variable zugeordnet wird. Diese soll die Werte aller Instanzen enthalten. Damit können beliebig viele Instanzen erzeugt werden, deren Werte des Slots *state* Felder der Länge 50 sind. Jedes dieser Felder wird in einem Prozessor gespeichert. Statt der Felddeklaration *'(array ...)* könnten hier beliebige andere PVAR-Typen (z.B. *string-char-pvar*) zur Anwendung kommen.

Die parallelen Slots werden mit speziellen Methoden verarbeitet, die für das Klassen-Metaobjekt *parallel-slot-class* bereitgestellt werden. Diese Definition entspricht damit der Schnittstelle eines CLOS-Programms zu *Lisp. Die weiteren Definitionen beschreiben ihre interne Verarbeitung, sie können einem CLOS-Programmierer daher verschlossen bleiben.

Wie und wo wird aber nun das neue Metaobjekt in das MOP eingefügt, um das gewünschte

Verhalten zu erreichen? Wir wollen für jeden parallelen Slot eine PVAR vom gegebenen Typ erzeugen und nicht für jede *Instanz* der Klasse *many-instances* eine PVAR. Informationen über alle Instanzen einer Klasse werden im Klassen-Metaobjekt gehalten. Definieren wir also eine neue Klassen-Metaobjektklasse:

```
(defclass parallel-slot-class (standard-class)
  (pvars-of-parallel-slots))
```

Im Slot *pvars-of-parallel-slots* soll pro parallelem Slot eine PVAR eingetragen werden. Da das Metaobjekt von *many-instances* als Instanz von *parallel-slot-class* repräsentiert ist, müssen bei der Erzeugung dieser Instanz die PVARS alloziert und in *pvars-of-parallel-slots* eingetragen werden. Dies geschieht bei der Definition von *defclass* von *many-instances*. Betrachten wir die Methodenreihenfolge des MOP, falls eine Klasse mit *defclass* definiert wird:

```
defclass
ensure-class
   make-instance
      allocate-instance
      initialize-instance
      initialize-instance :after
   make-instance :after
```

Durch die obige Definition von *many-instances* mit *defclass* wird letztlich durch *allocate-instance* eine Instanz von *standard-class* erzeugt, mit folgenden Slots:

```
class       <pvar-slot-class>
slots       leerer Speicher für Slotwerte
```

Man beachte, daß die generische Funktion *allocate-instance* der Klasse *standard-class* und nicht der Klasse *pvar-slot-class* aufgerufen wird, da über die Klasse von *pvar-slot-class* diskriminiert wird. In *slots* wird der Speicher für die Einträge eines Klassen Metaobjekts bereitgestellt. Neben den von *standard-class* geerbten slots (u.a. *name*, *direct-slots*, vgl. Abbildung 1) wird auch das neue Slot *pvars-of-parallel-slots* berücksichtigt. In *initialize-instance* werden diese Einträge mit den Werten belegt (d.h. *'many-instances*, ((state :allocation ...)) usw.). Man erhält als Metaobjekt:

```
class       <pvar-slot-class>
slots       many-instances
            ((state :allocation ...))
            ()
```

Mit einer noch leeren Liste für die parallelen Variablen.

In *initialize-instance :after* schließlich werden die für *Klassen Metaobjekte* spezifischen Initialisierungen durchgeführt. Das sind u.a. die Erzeugung von Slotdefinitionen aus den Listen und die Ermittlung der Klassenpräzedenzliste:

```
class       <pvar-slot-class>
slots       many-instances
            ((slot-definition of state))
            ()
```

In der *initialize-instance :after* Methode ist damit das Klassen-Metaobjekt bereits vorhanden, hier kann also die gewünschte PVAR erzeugt und eintragen werden (*:after*-Methoden werden alle ausgeführt, *call-next-method* ist daher hier nicht notwendig):

```
(defmethod initialize-instance :after ((class parallel-slot-class)
                                       &rest initargs)
  (setf (slot-value class 'pvars-of-parallel-slots)
        (mapcar #'(lambda (slot)
                    (cons (slot-definition-name slot)
                          (allocate-pvar-for-slot slot)))
                (remove-if-not #'pvar-slot-p
                               (class-direct-slots class)))))
```

Für alle Slots, die eine PVAR verlangen, wird diese in *allocate-pvar-for-slot* unter Berücksichtigung des PVAR-Types erzeugt (dies soll hier nicht weiter betrachtet werden). Der Metaobjektslot *pvars-of-parallel-slots* wird mit den erzeugten Paaren *(slot-name pvar)* gefüllt.

Damit wird für jeden parallelen Slot eine parallele Variable vom gewünschten Typ erzeugt und im Klassen-Metaobjekt gespeichert. Die parallele Variablen stellen Speicherplatz für die Slotwerte zur Verfügung, ihre Länge sei der einfachheithalber hier als ausreichend zu betrachten. (Tatsächlich kann die CM beliebig lange PVARS in virtuellen Prozessoren bearbeiten, deren Anzahl über die der physikalischen Prozessoren hinausgeht.) In jedem Prozessor kann mit der obigen Definition ein Wert vom Typ *'(array (unsigned-byte 16) 50)* gespeichert werden.

Der nächste Schritt bezieht sich auf die Erzeugung von Instanzen der Klasse *many-instances*. Dabei sollen die angegebenen Initialwerte in je einen Prozessor eingetragen werden und dieser als aktiv gekennzeichnet werden. (Die später auszuführenden Operationen betrachten nur aktive Prozessoren also der Anzahl der Instanzen entsprechend viele.) Die Initialwerte werden aus *initialize-instance* mit *(setf slot-value-using-class)* gesetzt. Diese Funktion bezieht sich wie ihre Kollegen *slot-value-using-class, slot-boundp-using-class, slot-makunbound-using-class* auf *einen* Slotwert und nicht auf die zugeordnete parallele Variable. Es wird daher festgelegt, daß diese Slotfunktionen für das Klassen-Metaobjekt *parallel-slot-class* den Slotwert des letzten gesetzten Prozessors der parallelen Variablen bearbeiten. (Andere Spezifikationen für weitere spezialisierte Klassen sind hier denkbar.) Dieser enthält die Slotwerte der letzten Instanz. Wir benötigen damit für jeden parallelen Slot einen Index des nächsten freien Prozessors in der zugehörigen parallelen Variablen. Dieser muß pro erzeugter Instanz inkrementiert werden.

Dies geschieht in *allocate-instance* von *parallel-slot-class*: Man beachte, daß hier mit *(make-instance #<parallel-slot-class MANY-INSTANCES>)* gestartet wird und daher gemäß der oben beschriebenen Methodenreihenfolge *allocate-instance* auf *parallel-slot-class* und nicht auf *standard-class* angewandt wird.

```
(defmethod allocate-instance ((class parallel-slot-class)
                              &rest initargs)
  (let ((instance (call-next-method)))
    (mapc #'(lambda (slot-pvar-pair)
              (increment-pvar-counter (cdr slot-pvar-pair)))
          (slot-value class 'pvars-of-parallel-slots))
  instance))
```

Dazu nehmen wir an, daß statt der PVAR eine Struktur mit PVAR und zugehörigem Index im *cdr* gespeichert ist. Da bereits der Speicherplatz für die parallele Variable bei der Definition der Klasse alloziert wurde (s.o. *allocate-pvar-for-slot*), sind hier keine weiteren Aktionen

notwendig. Die PVAR wird damit bei der Definition mit *defclass* erzeugt und die Prozessoren durch den verwendeten Zähler aktiviert. Andere Vorgehensweisen sind hier denkbar, da die CM z.B. auch dynamische Allozierung erlaubt.

Die obengenannten Slotfunktionen sollen auf den letzten Prozessor zugreifen, falls ein paralleler Slot vorliegt. Man erhält:

```
(defmethod (setf slot-value-using-class) ((class parallel-slot-class)
                                          instance slot-name)
  (let ((slot (find slot-name (class-slots class)
                    :key #'slot-definition-name)))
    (if (and slot (pvar-type-p slot))
        (write-pvar-slot-value new-value
          (get-pvar-definition-for-slot class slot-name))
       (call-next-method))))
```

Die anderen Funktionen sind analog (vgl. auch [Kiczales et al., 1991, 97]). [2] *get-pvar-definition-for-slot* ermittelt die PVAR des Slots, in diese wird in *write-pvar-slot-value* mit *pref* (s.o.) und dem Index der Wert gespeichert.

Durch die bisherige Spezifikation von *parallel-slot-class* ist es möglich, über *(make-instance 'many-instances :state value)* Instanzen in parallel arbeitende Prozessoren zu speichern. Um die erzeugten parallelen Variablen zu verwenden, müssen Methoden für die Metaobjektklasse *pvar-slot-value* definiert werden. Sie werden mit dieser Klasse dem CLOS-Programmierer zur Verfügung gestellt. Als Beispiel kann man sich eine generische Funktion vorstellen, die das Maximum der Slotwerte berechnet:

```
(defgeneric maximum-instance (class slot))

(defmethod maximum-instance ((class parallel-slot-class) pvar-slot)
      <passende *Lisp-Funktion>
)
```

Da ein Wert bzgl. aller Instanzen von *many-instances* berechnet werden soll, wird die Funktion mit dem Klassen-Metaobjekt dieser Klasse aufgerufen:

```
(maximum-instance (find-class 'many-instances) 'state)
```

Alternativ dazu kann mit:

```
(defmethod maximum-instance ((instance many-instances) slot)
  (maximum-instance (class-of instance) slot))
```

und *(maximum-instance (make-instance 'many-instances) 'state)* über jede einzelne Instanz, auf alle Slotwerte der bisher erzeugten Instanzen zugegriffen werden.

Die Definition von generischen Funktionen, die auf den PVARS der hier definierten Metaobjektklasse arbeiten, wird zukünftig untersucht werden. Sie bilden letztendlich die Schnittstelle von CLOS zu *Lisp.

[2] Die Verwendung von *slot-name* entspricht der Beschreibung in [Kiczales et al., 1991]. Vorgeschlagen wird allerdings stattdessen *slot-descriptor* einzusetzen, was eine grössere Variabilität erlaubt.

4 Zusammenfassung

Das MOP bietet die Möglichkeit die Struktur und die Verarbeitung diverser Konzepte von CLOS (z.B. Klassen, Slots) mittels Metaobjektklassen neu zu spezifizieren. Es wurde untersucht, wie spezialisierte Metaobjektklassen zu definieren sind, um die Rechenleistung paralleler Maschinen (hier der Connection Machine) auszunutzen. Dazu wurde die Spezifikation einer Metaobjektklasse beschrieben, die es erlaubt die Werte bestimmter Slots über alle Instanzen parallel zu verarbeiten. Dabei wurde nur die Schnittstelle zum MOP betrachtet, die Implementation der CM-bezogenen Funktionen wurde aus Platzgründen nicht verdeutlicht.

Es zeigte sich, daß nur geringfügige Anpassungen notwendig sind, um die erwähnten Erweiterung von CLOS zu implementieren. Schwieriger gestaltete sich die *Ermittlung* der generischen Funktionen des MOP, deren Funktionalität erweitert werden sollte. Dazu wurde die Reihenfolge der generischen Funktionen des MOP näher untersucht.

Gegenstand weiterer Untersuchungen ist die Definition von generischen Funktionen auf den parallelen Datenstrukturen. Werden gemäß dieser Art noch andere spezielle Metaobjektklassen hinzugefügt, erhält man eine (variable) Sammlung von Hilfsmitteln, die es zum einen erlauben soll, gegebene CLOS-Programme durch nur kleine Änderungen auf der Connection Machine laufen zulassen. Zum anderen wird durch die Metaobjektklassen eine höhere Abstraktionsebene eingeführt, die die interne Verarbeitung komplexer Strukturen verdeckt.

Literatur

Annot, J.K. und P.A.M den Haan (1988), *POOL and DOOM: The object oriented Approach*, in P.C. Treleaven, Parallel Computers, Object-Oriented, Funcional, Logic, Wiley & Sons, 47-79.

Anthonisse, J. M., 1973, *A Graph-Defining Language*, Amsterdam Mathematisches Centrum, Report 30/73.

Bobrow, D.G. und G. Kiczales, 1988, *The Common Lisp Object System metaobject kernel: A status report.*, In Conference on Lisp and Functional Programming.

Boley, H., 1980, *Processing directed recursive labelnode hypergraphs with FIT programs*, Universität Hamburg, Report IFI-HH-M-81/80.

Boley, H., 1991, *Declarative Operations on Nets, A Sampler of Relational/Functional Definitions*, DFKI, Kaiserslautern, Report RR-90-12.

Dewar, A., J. Schonberg, A. Schwartz und D. Dubinsky, 1986, *Higher Level Programming: Introduction to the Use of the Set-Theoretic Programming Language SETL.* Springer-Verlag.

Friedman, D. P., 1968, *GRASPE, Graph Processing: A LISP Extension*, University of Texas at Austin, Report TNN-84.

Keene, S.E., 1989, *Object-Oriented Prgramming in Common Lisp.*, Addison-Wesley Publishing Company.

Kiczales, G. und Bobrow, D.G., 1988, *The Common Lisp Object System specification: Metaobject protocol.*, Technical Report 88-003, X3J13 standards committee document.

Kiczales, G., Bobrow, D.G. und des Rivières, J., 1991, *The Art of the Metaobject Protocol*, The MIT Press, Cambridge, Massachusetts.

*Lisp, 1988, **Lisp Reference Manual, Version 5.0*, Thinking Machines Corporation, Cambridge, Massachusetts.

Paepcke, A., 1990 *PCLOS: Stress Testing CLOS, Experiencing the Metaobject Protocol*, Proc. OOPSLA '90, ACM Sigplan 25, Vol. II, 194-211.

Rheinboldt, W.C. und V.R. Basili, 1972, *On a Programming Language for Graph Algorithms*, BIT, 12, 220-241.

Steele, G.L. Jr. und G.J. Sussman, 1978, *The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two)*, AI Memo 53, MIT, Artificial Intelligence Laboratory.

Treleaven, P.C., 1988, *Parallel Computers, Object-Oriented, Functional, Logic*, Wiley & Sons.