Künstl Intell (2012) 26:69–74
DOI 10.1007/s13218-011-0150-7

PROJEKT

Verwendung von LISP in KI-Projekten

Christian Betz · Lothar Hotz

Online publiziert: 17. November 2011

Zusammenfassung In diesem Beitrag werden anhand von anwendungsorientierten Forschungsprototypen aus den Bereichen Konfigurierung, Diagnose und Szeneninterpretation wesentliche LISP-Merkmale vorgestellt. Weiterhin wird ausgehend von den gemachten Erfahrungen eine Brücke zu aktuellen Entwicklungen aus dem Bereich der funktionalen Programmiersprachen geschlagen.

Schlüsselwörter Common Lisp · Wissensbasierte Systeme · Diagnose · Konfigurierung · Szeneninterpretation

1 Einleitung

In den letzten Jahren wurde LISP von uns in verschiedenen anwendungsorientierten KI-Projekten eingesetzt. Dabei wurden in der Regel Prototypen erstellt, die eine Bandbreite von Funktionalitäten aufweisen. Es entstanden heterogene Systeme, die Module unterschiedlicher Programmiersprachen integrieren. So konnte gezeigt werden, wie forschungsrelevante KI-Verfahren für industrielle Lösungen eingesetzt werden können.

Die erfolgreiche Verwendung von LISP in diesen Projekten beruht auf der besonderen Vielfalt an programmiertechnischen Möglichkeiten, die LISP bietet. Dazu gehört CLOS (das COMMON LISP Object System) mit seinem Meta-Object Protocol, kurz MOP [15], welches den Zugriff auf die Meta-Ebene der Objektorientierung erlaubt. Ein weiteres Merkmal liefern Makros; sie erlauben es zur Übersetzungszeit den vollen Umfang von LISP für die Transformation von beliebigen syntaktischen Strukturen in LISP-Code zu verwenden. In den Fallstudien in Abschn. 2 werden u.a. diese Merkmale verwendet, um modulare Entwicklungen durchzuführen und wissensbasierte Sprachen zu spezifizieren. In Abschn. 3 gehen wir genauer auf wesentliche LISP-Merkmale ein und detaillieren ihre Verwendung in den Fallstudien. In Abschn. 4 ziehen wir Schlüsse für zukünftige Entwicklung mit LISP und ähnlichen Sprachen.

C. Betz
Playmaker Studio GmbH, Hamburg, Deutschland
e-mail: chris@playmakerstudio.com

L. Hotz (✉)
HITeC e.V., Universität Hamburg, Hamburg, Deutschland
e-mail: hotz@informatik.uni-hamburg.de

2 Fallstudien mit COMMON LISP

In den Fallstudien wurde LISP für die Entwicklung von vollfunktionsfähigen Forschungsprototypen verwendet, welche später als Vorlage für die Entwicklung von Produkten und Weiterentwicklungen herangezogen wurden (vgl. Boxen 1–3). Die entwickelten Systeme realisieren Ansätze aus den KI-Bereichen Konfigurierung (Fallstudie 1), Diagnose (Fallstudie 2) und Szeneninterpretation (Fallstudie 3).

Die Fallstudien haben folgende gemeinsame Merkmale:

Gemischte Bottom-up- und Top-down-Entwicklung Zu Beginn der Entwicklung der Prototypen war u.a. nicht klar, welche Komponenten, Algorithmen, Datenstrukturen für die Realisierung der Ansätze notwendig sein werden. Daher lag naturgemäß auch keine detaillierte Anforderungsspezifikation vor. Eine agile Entwicklungsmöglichkeit, wie sie LISP liefert, gewährleistet kurzfristigen Responz zu ersten Realisierungsideen und so auch eine Detaillierung der Anforderungen. Hierdurch wird ein gemischtes Vorgehen ermöglicht, welches generelle Konzepte durch ständiges Überprüfen mittels ersten Implementierungen verifiziert.

Fallstudie 1 Konfigurierung mit PLAKON, KONWERK und ENGCON

Konfigurierung bezeichnet die Zusammenstellung einer Beschreibung eines Produkts, welches aus parametrisierbaren Komponenten besteht, so dass das Produkt bestimmte Aufgaben erfüllen kann [4]. Konfigurierungswerkzeuge liefern dabei die domänenunabhängigen Inferenzmechanismen (z.B. Regelbasierte Verfahren, Constraint-Löser, Beschreibungslogik) mit ihren Wissensrepräsentationssprachen, die es erlauben, die Komponenten, Relationen und Restriktionen einer Domäne zu beschreiben. Durch die Vielzahl von Komponenten und Restriktionen wird gerade in komplexen Domänen, wie z.B. die Konfigurierung von Antriebssystemen [19], eine Kombination von Inferenzverfahren benötigt. Das LISP-basierte Werkzeug KONWERK (entwickelt 1991–1995, [11]) wurde auf der ebenfalls LISP-basierten Implementierung PLAKON (entwickelt 1985–1990, [5]) aufgesetzt. Dabei wurde CLOS und das MOP verwendet, um die objektorientierte Sprache so zu erweitern, dass eine komponentenorientierte Wissensrepräsentationssprache entsteht [13]. Von PLAKON wurde die Realisierung von Komponentenbeschreibungen in einer Begriffshierarchie und die damit verbundenen Inferenzen (wie z.B. Test der Klassenzugehörigkeit einer Instanz) übernommen. Um die unterschiedlichen Inferenzverfahren anwendungsspezifisch kombinieren zu können, wurde eine erweiterbare Modulstruktur entwickelt, die im Wesentlichen auf der dynamischen Erweiterbarkeit von CLOS-Klassen beruht.
KONWERK wurde für die Entwicklung einer kommerziellen Re-Implementation in JAVA als Vorlage verwendet (ENGCON [12]). Die Entwicklung wurde in JAVA und nicht in LISP durchgeführt, weil auf eine weiter verbreitete Sprache zurückgegriffen werden sollte.

Fallstudie 2 Diagnose mit D3

Diagnosesysteme sind eine Klasse von Softwaresystemen, die aus einer gegebenen Menge von Merkmalen eines Falls auf eine relevante Untermenge aus der vorgegebenen Gesamtmenge der möglichen Diagnosen schließen. Dazu gibt es verschiedene Problemlösungsverfahren wie Regelbasiertes Schließen, Fallbasiertes Schließen, Bayessche Netze oder Neuronale Netze. Das LISP-basierte D3 [18] (mit den entsprechenden Vorläufern MED und MED2) ist ein sogenannter Expertensystemshellbaukasten für die Entwicklung von Diagnosesystemen. Fachexperten können mit Hilfe graphischer Editoren die Merkmals- und Diagnosemengen eingeben sowie das für die jeweiligen Problemlöser notwendige Wissen, z.B. Regeln, Wahrscheinlichkeiten oder Gewichte.
Als Baukasten ist D3 die Grundlage für eine ganze Reihe von Diplomarbeiten und Dissertationen gewesen, die jeweils unterschiedliche Aspekte in eigenen Modulen entwickelt haben, so beispielsweise ein fallbasiertes intelligentes Trainingssystem oder ein graphischer Klasseneditor (ein Vorläufer modellbasierter Entwicklung). Die Verwendung von LISP als Programmiersprache erlaubte es, in der überaus komplexen Domäne schnell zu ersten Prototypen zu kommen und dank des interaktiven Charakters Fehler in produktiven Systemen schnell zu erkennen und zu beseitigen. Die Wissensbasen wurden in einer domänenspezifischen Sprache (DSL) gespeichert und konnten so bei Bedarf auch manuell editiert werden; eingelesen wurden sie über den LISP-Reader. Eine Re-Implementierung in JAVA übernahm die Erfahrungen bei der Entwicklung von D3, um mit d3web (www.d3web.de) ein System zu entwickeln, das durch eine „Trendsprache“ mehr Studenten anzieht.

Komplexe Systeme Um die wesentlichen Merkmale der Ansätze zu demonstrieren, waren komplexe Systeme notwendig, nicht nur einzelne Komponenten. Dabei wurden verschiedene Modellierungs- und Inferenztechniken gekoppelt und an Anwendungsszenarien getestet. Hierdurch wurden neben algorithmischen Aspekten auch Integration, Datenstrukturen und Schnittstellen betrachtet. Die Entwicklung unterschied sich daher stark von anderen Projekten, in denen einzelne Komponenten entwickelt werden.

Keine Standardlösung Die Ansätze konnten nicht mit vorhandenen Konzepten, Implementierungen oder Referenzsysteme realisiert werden. Stattdessen war eine Entwicklung „from scratch“ notwendig, was nicht heisst, dass nicht einzelne Komponenten anderer Systeme (wie z.B. die Wissenmodellierung) wieder verwendet werden konnten[1]; teilweise bauen die Fallstudien aufeinander auf. Für die Erstellung der Prototypen war es in allen Fällen nicht zwingend auf „moderne“ Sprachen zurückzugreifen, da der wissenschaftliche Betrieb (auch in Anwendungsprojekten) diesbezüglich mehr Freiheiten lässt als ein kommerzielles Umfeld, wo beispielsweise andere Lizenzen greifen. Für die Re-Implementationen wurden allerdings Sprachen wie JAVA gewählt; die Gründe hierfür liegen eher im universitären Entwicklungsumfeld als an LISP selbst:

[1] Dies wurde auch durch die frühe Standardisierung von COMMON LISP [1, 20] möglich, die eine kontinuierliche Wiederverwertung eigener und anderer Module über einen langen Zeitraum hinweg (von 1985 bis jetzt) erlaubte.

Fallstudie 3 Szeneninterpretation mit SCENIC und SCENIOR

Szeneninterpretation ermöglicht die Erkennung von Objekten und Vorgängen in Bildern und Videos. Auf der Basis von KONWERK wurde das High-Level Szeneninterpretationssystem SCENIC entwickelt (von 2001–2005, [14]), welches es ermöglicht Standbilder und Videos zu interpretieren. Dazu werden von einem Bildverarbeitungssystem ermittelte Ansichten von Objekten zu 3D-Objekten und diese wiederum zu zusammengesetzten Objekten (Aggregaten) und Vorgängen kombiniert. Diese Kombination wurde bei SCENIC als Konfigurierungsaufgabe gesehen, so dass das Konfigurierungswerkzeug als solches verwendet werden konnte und nur entsprechende Modelle für die Bildverarbeitungsdomänen (hier Häuserfassaden und Tischdeckszenen) entwickelt werden mussten. Für die Wissensrepräsentation wurde eine auf Makros basierende Modellierungssprache entwickelt, die auf das in LISP implementierte Constraint-System von KONWERK abbildet. Für die Kombination des LISP-Moduls mit anderen in C++ und Python realisierten Bildverarbeitungsmodulen wurde eine XMLRPC-Schnittstelle (www.xmlrpc.com) verwendet. Diese Schnittstelle ermöglicht den Aufruf von Methoden unterschiedlicher Programmiersprachen, ohne dass ein in LISP-Implementationen teilweise vorhandenes, nicht standardisiertes Foreign-Function-Interface benutzt werden muss.
SCENIC wurde als Vorlage für die Entwicklung von SCENIOR (von 2006–2010, [3]) herangezogen. SCENIOR ist eine auf JAVA und dem regelbasierten Werkzeug JESS [8] basierende neue Implementierung eines Szeneninterpretationssystems. Die Verwendung von JESS mit der Möglichkeit der Realisierung mehrerer paralleler, alternativer Szeneninterpretationen war ein wesentlicher Grund für die Neuentwicklung des Systems. Die Parallelisierung wird hierbei durch eine Klonmöglichkeit der Regelbasen und Arbeitselemente (working-element) des Regelsystems realisiert; d.h. die Inferenzmaschine (das Regelsystem) beinhaltet bereits die Parallelisierungsmöglichkeit. Diese Funktionalität war so in KONWERK nicht vorhanden.

Es sollte mit JAVA die Sprache verwendet werden, die die Studierenden auch in den übrigen Kursen verwendeten.

3 Merkmale von LISP

Was macht LISP anders als andere Sprachen? Warum ist LISP auch über 60 Jahre nach seiner Entwicklung durch McCarthy [16] noch eine Sprache, die zu den Top 20-Programmiersprachen zählt (vgl. Abschn. 4.1)? Es sind die typischen Eigenschaften von LISP, die es so besonders machen (vgl. auch [2, 10]):

3.1 LISP-Programme sind nicht nur Programme, sondern auch Daten

Das auffallenste Merkmal von LISP ist die durchgehende Verwendung von Listen. Listen kommen dabei einzigartiger Weise in zwei Rollen vor, nämlich als Datenobjekte, z.B. die Liste ′(1 2 3), und als Programme (Beispiel s.u.). Dieses Phänomen ermöglicht speziell das im Folgenden beschriebene Makrosystem, welches Programme (also Listen als Programme) mittels Listenfunktionen (also Listen als Daten) manipuliert. Bemerkenswert ist auch die einfache Lesbarkeit und Handhabbarkeit von LISP-Programmen. Nach kurzzeitiger Eingewöhnung werden Klammern, welche die Listen umschließen, nicht mehr wahrgenommen und es entsteht vor dem geistigen Auge des Programmierers ein PYTHON-ähnliche Visualisierung des Programms (vgl. auch [10], S. 18). Wesentlicher ist jedoch die Möglichkeit mittels Code-Snippets (d.h. verschachtelter Listen) Programm-Code zusammenzusetzen. Während der Entwicklung mit LISP erkennt man: Listen bilden kein Hindernis, sondern liefern eine effektive Unterstützung.

3.2 LISP ist funktional

Das bedeutet zunächst, dass LISP-Programme aus Funktionen bestehen, die in der Regel seiteneffektfrei sind und die daher bei gleichen Eingabeparametern immer das gleiche Ergebnis liefern. Es bedeutet aber auch, dass Funktionen sogenannte *first-class objects* in der Sprache sind, also genauso verwendet werden können wie andere Objekte, etwa Zahlen oder Zeichenketten.

Warum ist das etwas Besonderes? Es vereinfacht das Entwickeln und das Testen von Software dramatisch. In einem funktionalen Programm kann jede Funktion für sich entwickelt und getestet werden. Da sie seiteneffektfrei ist, muss nur der Rückgabewert überprüft werden, ein Überprüfen der „Umwelt“ ist nicht notwendig. Im Zusammenspiel mit der REPL, der *Read-Eval-Print-Loop* (siehe unten), können Funktionen so auch interaktiv entwickelt werden. Diese Stärke spielt LISP natürlich besonders mit Test-Driven-Development bzw. Behaviour-Driven-Development aus, indem die Tests in der Regel kürzer und damit verständlicher sind als in Sprachen, die stark auf Seiteneffekten, z.B. der Veränderung von Objekten, beruhen.

Natürlich lassen sich „real life“-Programme nicht rein funktional entwickeln – eine Ausgabe auf dem Bildschirm beispielsweise ist ein Seiteneffekt. Der typische Stil bei der

Entwicklung von LISP-Programmen ist es allerdings, einen möglichst großen Anteil der Software rein funktional zu entwickeln.

In den Fallstudien beispielsweise ermöglichten first-class Funktionen kompakten Code, indem z.B. für das Sortieren, Suchen oder Löschen in Listen Test- und Zugriffsfunktionen direkt beim Aufruf übergeben werden können. Im folgenden Beispiel wird die Löschfunktion `remove-if-not` mit der Testfunktion `is-even` aufgerufen:

```
Definitionen:
(defun is-even (number)
  (= (mod number 2) 0))
(defun even-numbers-only (list-of-numbers)
  (remove-if-not #'is-even list-of-numbers))
Aufruf:   (even-numbers-only '(1 2 3 4 5))
Ergebnis: (2 4)
```

In Sprachen, die diese Funktionen nicht anbieten, muss hier beispielsweise mit einem Strategie-Pattern gearbeitet werden. Dies ist sehr typisch für LISP, dass Probleme mit Sprachmitteln gelöst werden, die in weniger ausdrucksstarken Sprachen die Verwendung eines Patterns erfordern.

3.3 LISP ist interaktiv und dynamisch

LISP ist interaktiv und stellt mit der *Read-Eval-Print-Loop*, kurz REPL, eine Art Kommandozeile zur Verfügung. In der REPL wird LISP-Code interaktiv ausgeführt, von einfachen ersten Schritten beim Erlernen von LISP über die Entwicklung eines neuen Programms bis hin zum direkte Einsprung in ein System auch zur Laufzeit.

Dank eines dynamischen Typsystems müssen Typinformationen nicht statisch vom Entwickler festgelegt werden, sondern werden zur Laufzeit automatisch bestimmt. So wird bei einer Funktionsdefinition weder der Typ der Parameter noch der Typ des Rückgabewerts festgelegt. Was zunächst fehlerträchtig klingt (keine Typüberprüfung durch den Compiler) macht den Code sehr viel übersichtlicher (kein „Boilerplate“ für etwas, das automatisch bestimmt werden kann). Es erlaubt auch die Reduktion von unnötigen Klassen in der Objektorientierung. In JAVA ist beispielsweise für das Compositum-Pattern eine (abstrakte) Basisklasse notwendig, deren einziger Sinn es ist, eine Liste von Objekten zu pflegen, deren Klasse von dieser abstrakten Basisklasse abgeleitet ist, und entsprechende Methoden aufzurufen, die aber nur von den Kindklassen implementiert werden.

LISP ist eine dynamische Sprache, das heißt, dass im laufenden System Änderungen möglich sind, z.B. durch die Re-Definition von Funktionen. Die Trennung zwischen Entwicklungs- und Laufzeit verschwimmt damit. In der JAVA-Welt heißt so etwas typischerweise „Hot-Code-Replacement“, also die Möglichkeit in der Laufzeitumgebung Code zu verändern. So kann man beispielsweise in einem laufenden LISP-Programm Funktionen neu definieren, um einen erkannten Fehler zu korrigieren und muss nicht das gesamte Programm neu starten. Teil dieses dynamischen Systems sind sogenannte *Restarts*, die es erlauben, bei Ausnahmesituationen aus dem laufenden Programm im Aufruf-Stack zurückzuspringen. Der Entwickler kann dann einen Fix entwickeln und mit der Ausführung des Systems direkt fortfahren (vgl. [17]). Durch diese direkt Präsenz eines Debuggers wird keinerlei Neuübersetzungen für detaillierte Debug-Zwecke notwendig. All dies sorgt dafür, dass sich Entwickler in LISP oft „näher am Code“ fühlen.

In den Fallstudien wurde die *Read-Eval-Print-Loop* und der dynamische Charakter von LISP für schnelle Tests und Entwicklungen eingesetzt. Die Dynamik der Sprache ermöglicht weiterhin die kontinuierliche Entwicklung ohne lange Übersetzungs- und Applikationsstartzeiten. Für D3 waren beispielsweise die – je nach Variante – mindestens 40 Minuten dauernden *Builds* nur sehr selten notwendig, die Zeit der Entwickler konnte so sehr viel sinnvoller genutzt werden.

3.4 Makros – „The whole language always available“ (Paul Graham, www.paulgraham.com/diff.html)

Das LISP-Makrosystem [9] ist eines der faszinierendsten Merkmale von LISP. Mittels dieser Makros kann man die Sprache selbst an das Problem anpassen, etwas, das sich weit nach der Einführung in LISP als Domain-Specific-Languages (DSL) durchgesetzt hat. Dabei gehen LISP-Makros weit über die Makrosysteme anderer Sprachen hinaus und erlauben tatsächlich die Erweiterung der Sprache selbst in einer Art, die vom „Kern“ nicht zu unterscheiden ist.

Die LISP-Makros erlauben es darüber hinaus, Code-Duplikation drastisch zu reduzieren und sind einer der Gründe dafür, warum – je nach Statistik und Lesart – gegenüber Programmiersprachen wie JAVA oder C/C++ nur ein Drittel bis ein Zehntel des Codes zur Lösung des gleichen Problems notwendig sind (vgl. [6]).

Makros sind das Ergebnis der Anwendung von LISP-Code zur Übersetzungszeit (*Compile-Time*), so dass man für die Makroprogrammierung den vollständigen Umfang von LISP zur Verfügung hat. Die Anwendung von LISP zur Ladezeit (*Read-Time*) erlaubt die Erweiterung der LISP-Syntax – die Baumstruktur eines XML-Dokuments könnte beispielsweise auch direkt über die LISP-Ladestrukturen geparst und gelesen werden. Laden zur Laufzeit (*Run-Time*) ermöglicht den Austausch von Baumstrukturen – wieder wie in XML. Compiling zur Laufzeit schließlich ist die Basis für die Dynamik der Sprache. Damit löst sich die Unterscheidung in Read-Time, Compile-Time und Run-Time weitestgehend auf.

In den Fallstudien wurden Makros für die Definition von Modellierungssprachen und ihre Abbildung auf interne Inferenzalgorithmen verwendet. Beispielsweise wurden in den

Sprachen modellierte Wissensaspekte auf Constraints abgebildet. In [21] werden auf ähnlich kompakte Art beschreibungslogische Modelle beschrieben.

3.5 LISP ist objektorientiert: CLOS und MOP

Objektorientierung bieten auch viele andere Sprachen, aber auch hier ist LISP oft das entscheidende Quäntchen weiter – das CLOS (Common LISP Object System) bindet Methoden beispielsweise nicht an Klassen, wie dies bei JAVA der Fall ist, sondern verwendet generische Methoden, die sehr flexibel ausgewählt (dispatcht) werden können – nach einer Klasse, nach mehreren Klassen oder auch nach konkreten Werten der Parameter. Die Methoden können außerdem – ähnlich wie bei Aspekt-orientierter Programmierung – auch später durch :before, :after oder :around-Methoden erweitert werden.

Das Meta-Object Protocol, kurz MOP [15], erlaubt, wie bereits in der Einleitung erwähnt, den Zugriff auf die Meta-Ebene der Objektorientierung. In LISP ist es beispielsweise möglich, die Klasse eines Objekts zur Laufzeit zu verändern. Denn warum sollte man beispielsweise für einen stillgelegten Account nicht auch eine entsprechende Klasse verwenden, für die es dann keine Methoden zu Änderungen an dem Account mehr gibt. In anderen Programmiersprachen müsste man dafür ein neues Objekt anlegen und alle Referenzen auf das ursprüngliche Objekt „umbiegen“ – in LISP bleibt das Objekt bestehen, verändert aber seine Klasse und sein Verhalten.

In den Fallstudien konnten durch das MOP Merkmale wissensbasierten Schließens, wie z.B. die Vererbung, direkt von CLOS verwendet und angepasst werden. So konnten wissensbasierte Konzepte oder Frames durch Klassen der objekt-orientierten Sprache dargestellt werden. In den Re-Implementationen mit JAVA mussten Merkmale wie z.B. die Vererbung neu implementiert werden, d.h. ein Frame war ein Objekt einer Klasse, keine Java-Klasse.

4 Ausblick

4.1 Aktuelle Situation von LISP

LISP schneidet in den Statistiken und Analysen, die für die Situation der Verwendung von Programmiersprachen herangezogen werden können, eher schlecht ab (vgl. Job-Angebote auf langpop.com, Projektanalysen auf www.ohloh.net oder der TIOBE Programming Community Index, (www.tiobe.com/index.php/paperinfo/tpci/Lisp.html, tpci)). Im tpci rangiert LISP im August 2011 auf Rang 14, verglichen mit den gut 19% für JAVA und 17% für C nimmt LISP mit 0,9% aber einen geringen Raum ein. In der eWeek [7] schafft es CLOJURE (s.u.) immerhin auf Rang 17, während LISP dort gar nicht vertreten ist.

Nach den explorativen Phasen der Entwicklung von KI-Komponenten in LISP liegen jetzt (Re-)Implementationen in anderen Sprachen als leicht integrierbare Komponenten für die unterschiedlichen Inferenzmechanismen vor (wie Constraint-Systeme, Regelsysteme oder Beschreibungslogik-Reasoner). Auch Modellierungssprachen wie OWL, RULE-ML oder SWLR werden häufig für Standard-KI-Aufgaben eingesetzt. Bei Neuentwicklungen von Modellierungssprachen oder domänenspezifischen Sprachen kommen die Repräsentationsmerkmale von LISP jedoch zum Tragen.

Verfügbar ist LISP über kommerzieller Anbieter (z.B. ALLEGRO COMMON LISP (www.franz.com), LispWorks (www.lispworks.com)), sowie freie Varianten (wie z.B. CLOZURE (www.clozure.com), CLISP (www.clisp.org), SBCL (www.sbcl.org)), die sich im Wesentlichen über ihre Compiler unterscheiden und so unterschiedliche Laufzeiten von Anwendungen bedingen.

4.2 Das LISP-Revival mit CLOJURE

CLOJURE ist ein – primär von Rich Hickey entwickelter – LISP-Dialekt (clojure.org), der auf der JAVA Virtual Machine läuft. Dabei schlägt CLOJURE eine Brücke in die JAVA-Welt, so dass bestehende JAVA-Frameworks problemlos weiterverwendet werden können. CLOJURE bietet unveränderliche, persistente Datenstrukturen, die eine funktionale Programmierung ohne Seiteneffekte favorisieren und dazu ein Software Transactional Memory System für eine einfache und korrekte Unterstützung von Multi-Core-Entwicklungen.

Wie andere LISP-Dialekte ist CLOJURE dynamisch und funktional und bietet ein mächtiges Makrosystem. CLOJURE verwendet nicht CLOS und MOP für die Objektorientierung, geht mit den Multimethoden aber in die gleiche Richtung und stellenweise sogar über CLOS hinaus.

Durch die Nähe zu JAVA findet CLOJURE zunehmend Verbreitung und wird in Projekten auch mit anderen interessanten Technologien kombiniert. So entwickelt z.B. freiheit.com mit CLOJURE Web-Apps auf der Basis der Google App Engine. Ein anderes Beispiel ist flightcaster.com, wo u.a. CLOJURE für die Zahlenverarbeitung einsetzt wird.

4.3 Ein Blick in die Zukunft

LISP fasziniert auch über 60 Jahre nach seiner Entwicklung noch Entwickler durch seine Eleganz und Mächtigkeit, auch wenn die (vermeintliche) Klammernhölle Einsteiger immer wieder abschreckt. Daher prognostizieren wir (zumindest mittelfristig), dass weiter eine kritische Masse an Entwicklern und kontinuierlich weiterentwickelte LISP-Projekte existieren werden, die eine Basis für die zukünftige Verwendung von LISP bilden, im Wesentlichen ausserhalb der KI.

Ein wesentlicher Aspekt dabei wird auch der Einsatz von CLOJURE sein, das gut mit anderen Sprachen und vor allem mit dem weit verbreiteten JAVA interagiert und, durch den gezielten Bruch mit ANSI COMMON LISP, interessante neue Ideen verwirklicht.

5 Zusammenfassung

Die besonderen Eigenschaften von LISP ermöglichen eine flexible und effektive Vorgehensweise bei dem anwendungsbezogenen Einsatz bekannter Ansätze und der Entwicklung neuer Ansätze in der KI. Die hier betrachteten Fallstudien zeigen diese Vorgehensweisen in unterschiedlichen KI-Bereichen. Neuere Programmiersprachen wie CLOJURE nehmen LISP-Konzepte auf und integrieren sie in gängige Entwicklungsumgebungen und Anwendungskontexte.

Literatur

1. American National Standard X3.226: ANSI common lisp (1994)
2. Barski C Land of lisp, learn to program in lisp, one game at a time! No starch press (2010). URL: landoflisp.com, www.scribd.com/doc/51875893/282/EPILOGUE
3. Bohlken W, Neumann B, Hotz L, Koopmann P (2011) Ontology-based realtime activity monitoring using beam search. In: Crowley JL, Draper B, Thonnat M (Hrsg) ICVS 2011. Lecture notes in computer science, Bd 6962. Springer, Heidelberg, S 112–121
4. Chandrasekaran B (1990) Design problem solving: a task analysis. AI Mag 1(4):59–71
5. Cunis R, Günter A, Strecker H (Hrsg) (1991) Das PLAKON-Buch, Informatik-Fachberichte, Bd 266. Springer, Berlin
6. Erann G (2000) Lisp as an alternative to Java. Intelligence 11(4):21–24
7. eWeek, www.eweek.com/c/a/Application-Development/Java-C-C-Top-18-Programming-Languages-for-2011-480790/ (2011)
8. Friedman-Hill E (2003) Jess in action: Java rule-based systems. Manning, Greenwich
9. Graham P (1994) On lisp: advanced techniques for common lisp. Prentice Hall, New York
10. Graham P (1996) ANSI Common Lisp. Prentice Hall, Upper Saddle River. URL: www.paulgraham.com
11. Günter A (1995) Wissensbasiertes Konfigurieren. Infix, St. Augustin
12. Hollmann O, Wagner T, Günter A (2000) EngCon: a flexible domain-independent configuration engine. In: Proc. ECAI-workshop configuration, Berlin, Germany, S 94
13. Hotz L (2009) Frame-basierte Wissensrepräsentation zur Konfigurierung, Analyse und Diagnose technischer Systeme. DISKI, Bd 325. Infix, St. Augustin
14. Hotz L, Neumann B (2005) Scene interpretation as a configuration task. Künstl Intell 3:59–65
15. Kiczales J, Bobrow DG, des Rivieres J (1991) The art of the metaobject protocol. MIT Press, Cambridge
16. McCarthy J, Abrahams PW, Edwards DJ, Hart TP, Levin MI (1962) LISP 1.5 programmer's manual, 2. Aufl., 15th printing, 1985. MIT Press, Cambridge
17. Prechelt L (2000) An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl. Computer 33(10):23–29
18. Puppe F (1998) Knowledge reuse among diagnostic problem-solving methods in the shell-kit d3. Int J Hum-Comput Stud S 49 627–649
19. Ranze K, Scholz T, Wagner T, Günter A, Herzog O, Hollmann O, Schlieder C, Arlt V (2002) A structure-based configuration tool: drive solution designer DSD. In: 14. conf innovative applications of AI
20. Steele GLJ (1984) COMMON LISP the language. Digital Press
21. Wessel M (2007) Flexible und konfigurierbare Software-Architekturen für datenintensive ontologiebasierte Informationssysteme. Logos-Verlag, Berlin

Christian Betz is Senior Software Architect at Playmaker Studio. He founded knowIT-Software in 2003 from his Ph.D. research project. He was Senior Consultant IT at SinnerSchrader, providing architectural expertise to Germans largest financial institutions. At Plath, he steered the development of a new product line for network analysis.

Lothar Hotz is Senior Researcher at the Hamburgs Informatics Technology Center (HITeC e.V.) located at the University of Hamburg. He has participated in several projects related to topics of configuration, constraints, diagnosis, scene interpretation, requirements engineering, parallel processing, and object-oriented programming languages.