Migration und Kompilation in Lisp: Ein Weg von Prototypen zu Anwendungen

Wolfgang Goerigk, Friedemann Simon

Christian - Albrechts - Universität zu Kiel

Arbeitspaket: Komplettkompilation nach C
Status: Entwurf
Dokumentbezeichner: APPLY/CAU/IV/2
Erstellt am: 30.03.93
Korrigiert am: 17.05.93

Synopsis

Lisp-basierte Anwendungen behalten oft den Charakter von Prototypen und weisen deshalb Defizite auf in den Bereichen: Wartbarkeit, klares Design und sparsamer Umgang mit Ressourcen. Der auf die Kompilation von Anwendungsprogrammen ausgerichtete Ansatz von APPLY bietet einen Weg zur Qualitätsverbesserung. Es entstehen von einem Lisp-System unabhängige Applikationen, d.h. übliche Objektfiles oder C-Programme, die mit Betriebssystemmitteln zu fertigen Programmen verbunden werden können. Dies ist eine wesentliche Voraussetzung für eine wirkliche Integration in ein DV-Umfeld. Zur Erzeugung von kompakten Anwendungen ist zudem ein Migrationsschritt erforderlich, der die dynamischen und hochflexiblen Sprachkonstrukte eliminiert. Praktisch bedeutet das eine gründliche Überarbeitung des Programms nach gewissen Migrationsregeln und allgemeinen Prinzipien des Software-Engineering. Nach vorliegenden Erfahrungen sind Migrationsschritte nur an wenigen Stellen erforderlich, wo man sich beim Prototypen mit einer ad-hoc-Lösung durch ein dynamisches, universell einsetzbares Sprachmittel beholfen hat. Durch statische Programmanalyse werden zudem Programmierfehler aufgedeckt.

Diese Arbeit ist im Rahmen des Verbundvorhabens APPLY entstanden, das vom Bundesminister für Forschung und Technologie gefördert wird. Die Projektpartner sind die Christian-Albrechts-Universität Kiel, die Gesellschaft für Mathematik und Datenverarbeitung mbH (GMD), das Fraunhofer Institut für Software- und Systemtechnik (ISST) und VW-GEDAS.

Christian-Albrechts-Universität Kiel
Institut für Informatik und Praktische Mathematik
Forschungsgruppe „Programmiersprachen und Übersetzerkonstruktion“
Preußerstr. 1-9, W–2300 Kiel 1
Fax: 0431/5661-43
Ansprechpartner: Dr. Friedemann Simon
Tel.: 0431/5604-25
E-mail: fs@informatik.uni-kiel.dbp.de

Gesellschaft für Mathematik und Datenverarbeitung mbH (GMD)
Institut für Angewandte Informationstechnik
Forschungsbereich Künstliche Intelligenz
Postfach 1316, W–5205 Sankt Augustin 1
Fax: 02241/14-2072
Ansprechpartner: Prof. Dr. Thomas Christaller
Tel.: 02241/14-2678
E-mail: thomas.christaller@gmd.de

Fraunhofer Institut für Software- und Systemtechnik (ISST)
Kurstr. 33, O–1086 Berlin
Fax: 030/20372-301
Ansprechpartner: Dr. Horst Friedrich
Tel.: 030/20372-342
E-mail: horst.friedrich@isst.fhg.de

VW-GEDAS
Pascalstr. 11, W–1000 Berlin 10
Fax: 030/39007999
Ansprechpartner: Dipl.-Ing. Jürgen Beste
Tel.: 030/39007-331
E-mail: beste%sun1@hwsw.gedas.de

Inhaltsverzeichnis

1 Einleitung

Der breiten industriellen Anwendung Lisp-basierter Expertensystemtechnik stehen verschiedene Hindernisse im Wege [KG93] [Weu93] [MBG90]. Ein Problemkreis ist die mangelnde technische und organisatorische Integrationsfähigkeit in das industrielle DV-Umfeld. Ein weiteres Problem entsteht, weil die klassischen Vorgehensweisen bei einer Systementwicklung häufig unberücksichtigt bleiben. Die Anwendungen behalten den Charakter von Prototypen und weisen deshalb Defizite auf in den Bereichen: Wartbarkeit, klares Design und sparsamer Umgang mit Ressourcen.

Der auf die Kompilation von Anwendungsprogrammen ausgerichtete Ansatz von APPLY [BCF+91] [GS91] bietet einen Weg zur Qualitätsverbesserung unter besonderer Berücksichtigung der genannten Kriterien.

Grundannahme ist eine klare Trennungsmöglichkeit von Entwicklung und Anwendung eines Programms. Fertige Programme benötigen die Flexibilität und den Leistungsumfang von Lisp nicht mehr in dem Maße, wie es während der Programmentwicklung notwendig ist. Anwendungsprogramme können durch die Elimination des nicht benötigten Leistungsumfangs nicht nur bedeutend kleiner, sondern auch effizienter werden. Es entstehen von einem Lisp-System unabhängige Applikationen, da die Kompilationsergebnisse übliche Objektfiles oder C-Programme sind, die mit Betriebssystemmitteln zu fertigen Programmen verbunden werden können. Das ist eine wesentliche Voraussetzung für die wirkliche, sowohl technische als auch organisatorische Integration in ein DV-Umfeld.

Da der volle Sprachumfang von Lisp unter dem Kriterium der sparsamen Nutzung von Ressourcen, insbesondere Speicher, nicht kompilierbar ist, müssen fertige Programme in der Regel in eine Teilsprache migriert werden, in der einige der hochflexiblen, dynamischen Sprachkonstrukte, die den *Entwicklungsprozeß* von Programmen beschleunigen, nicht mehr auftreten. Im Rahmen von APPLY wurde zu diesem Zweck die Teilsprache CommonLisp_0 [Knu92] definiert. Praktisch bedeutet das eine gründliche, teilweise automatisierbare Überarbeitung des Programms nach gewissen Migrationsregeln und allgemeinen Prinzipien des Software-Engineering im Dialog mit dem Compiler, der gleichzeitig eine Analyse der statischen Semantik des Programms vornimmt und dabei Programmierfehler aufdecken kann. Der Programmieraufwand für die Migration ist im Vergleich zu einer Reimplementation z. B. in C gering, und das Vorgehen ist wesentlich weniger fehleranfällig.

Nach vorliegenden Erfahrungen sind Migrationsschritte nur an wenigen Stellen eines Programms erforderlich, wo man sich beim Prototypen mit einer ad hoc Lösung durch ein dynamisches, universell einsetzbares Sprachmittel beholfen hat. Die Erfahrungen stammen aus der Migration und erfolgreichen Kompilation von Anwendungen, die in Common Lisp [Ste84] [Ste90] entwickelt wurden, in die im Rahmen des APPLY-Projektes entstandene strikte Teilsprache CommonLisp_0 von Common Lisp.

Die vorliegende Arbeit stellt das methodische Vorgehen, Möglichkeiten und Grenzen des Ansatzes sowie Resultate über Migrationen und Kompilationen von Common-Lisp-Anwendungen externer Partner des APPLY-Projektes vor. Wir setzen die Kenntnis von Common Lisp [Ste84] [Ste90] voraus, die für das Verständnis des durchgängig benutzten Beispiels nötig ist.

Ziel der Migration ist ein Programm der Sprache CommonLisp_0, das über ein Hauptprogramm verfügt und sich somit als eigenständige Applikation mit dem im Rahmen des

Projektes APPLY entwickelten Komplettcompiler entweder als Ganzes oder modulweise in ein C-Programm und dann weiter in ein vom Lisp-System unabhängiges Maschinenprogramm übersetzen läßt. Dabei ist die Transformation von Lisp nach C und die dann anschließende Wartung und Weiterentwicklung des entstandenen C-Programmes ausdrücklich nicht ein Hauptziel dieser Arbeit. Vielmehr entsteht durch die Übersetzung C-Code, der eher dem einer abstrakten C-Maschine gleicht und nicht im Hinblick auf Lesbarkeit und Änderbarkeit, sondern im Hinblick auf Effizienz und Portabilität generiert wurde.

2 Ansatz und Aufgabenstellung

Diese Arbeit stellt einen Ansatz vor, wie Anwendungsprogramme in Lisp unter Ausnutzung des vollen Leistungsumfanges inkrementeller Lisp-Systeme entwickelt und danach effizient, portabel und integrationsfähig implementiert werden können. Grundannahme ist, daß die Entwicklung eines Programmes von dessen Anwendung klar zu trennen und in der Anwendung nicht mehr der volle Leistungsumfang des Entwicklungssystems nötig ist.

2.1 Programmentwicklung

Die Entwicklung eines Lisp-Programmes geschieht in der Regel in einem interaktiven Common-Lisp-System, häufig im Sinne von *rapid prototyping*, und unter Ausnutzung des vollen Leistungsumfangs von Common Lisp. Der schnelle Entwicklungszyklus wird dabei durch das interaktive System selbst, durch die in einem konkreten System zusätzlich bereitgestellte Funktionalität und auch durch die der Sprache Lisp eigenen flexiblen und dynamischen Sprachkonstrukte unterstützt. Insbesondere der in einem interpretierenden System mögliche Test unvollständiger Programme spielt dabei eine wichtige Rolle.

Wir halten dies für ein Leistungsmerkmal von Lisp und plädieren ausdrücklich nicht für eine Programmentwicklung in der bereits im letzten Abschnitt erwähnten Sprache CommmonLisp$_0$. Viele Lisp-Anwendungen, aber auch viele in Lisp-Anwendungen benutzte Programmiertechniken sind durch „Experimente in Lisp“ entstanden, und dies stellt eine eigene Qualität dar, die nicht aufgegeben werden sollte.

Wir hoffen andererseits, daß der aus softwaretechnischer Sicht nötige und sich an die Programmentwicklung anschließende Migrationsschritt letztlich auch *normativ* auf die Programmentwicklung auswirkt. Genauer: Dort, wo der Programmentwicklungsprozeß durch die rechtzeitige Verwendung übersetzbarer oder effizienterer Datenstrukturen und Algorithmen nicht gehemmt würde, sollten diese auch verwendet werden.

2.2 Migration

Analysiert man vorhandene Lisp-Applikationen genauer, so stellt man fest, daß sie an verschiedenen Stellen unverwechselbar den Charakter von Prototypen aufweisen:

- Sie enthalten häufig kein Hauptprogramm, sondern bestehen nur aus einer Reihe von Deklarationen, und ein Programmablauf besteht aus dem mehr oder weniger koordinierten und dokumentierten Aufrufen der zur Verfügung gestellten Funktionalität in einem interaktiven Lisp-System. Es sei an dieser Stelle darauf hingewiesen, daß manche in Lisp implementierten Systeme ihrem Wesen nach den Charakter von Spracherweiterungen von Lisp haben, die im Zusammenhang mit einem inkrementellen Lisp-System zur Programmentwicklung genutzt werden sollen. Diese Systeme selbst sind keine Lisp-Applikationen in dem hier verwendeten Sinne. Erst ihre Anwendungen würden wir als Lisp-Applikationen bezeichnen.

- Sie enthalten eine Reihe von Programmteilen, die den Entwicklungsprozeß unterstützen, die für einen Programmablauf aber völlig irrelevant sind. Dies sind z.B. Funktionen zum automatischen Rekompilieren, zum automatischen Nachladen oder zur Konsistenzerhaltung des Programmes, falls nur Teile redefiniert oder nachgeladen werden.

- Sie enthalten eine Reihe von Mehrfachdefinitionen derselben Funktion oder Prozedur, die in der Entwicklung der Reihe nach entstanden sind und von denen in einer inkrementellen Entwicklungsumgebung jeweils nur die als letzte in das System geladene die gültige ist.

- Sie enthalten eine Reihe von Programmpfaden und auch Funktionsdefinitionen, die nur zu Testzwecken entstanden sind und die in einem tatsächlichen Programmlauf nicht aufgerufen werden. Neben totem Code, den jeder gute Compiler eliminieren kann, sind dies aber auch Programmteile, die zwar formal aufgerufen, tatsächlich jedoch nie erreicht werden.

- Sie enthalten eine Reihe von Programmiertricks unter Ausnutzung sehr flexibler hochdynamischer, häufig reflektiver Lisp-Konstrukte, die im Entwicklungsprozeß zugunsten der schnellen Entwicklung durchaus ihre Berechtigung hatten und dann im Programm verblieben sind.

- Sie nutzen häufig Funktionalität, die nicht zur Sprache Lisp, wohl aber zu dem in einem konkreten Lisp-System bereitgestellten Funktionsumfang gehört.

Man stellt also fest, daß die Programme ohne Änderungen nicht den erforderlichen Einschränkungen genügen, die für Programme einer effizient kompilierbaren Teilsprache von Common Lisp verlangt werden müssen.

Vor der Kompilation ist daher ein weiterer Schritt nötig, den wir als *Migration* bezeichnen. Unter diesem Begriff verstehen wir eine lokale oder globale Sourcecode-Transformation von Programmen in einer Sprache in Programme in einer anderen (in unserem Fall einer Teilsprache der ersten) unter Erhaltung der Funktionalität des ursprünglichen Programms. Der Begriff läßt sich zwischen automatischer, semantikerhaltender Programmtransformation und Reimplementation einordnen. (vgl. Abbildung 1)

Dieser Migrationsschritt entspricht hier seinem Wesen nach einer manuellen Transformation, die durch das Entwicklungssystem und auch den Übersetzer unterstützt wird. Der Grund ist, daß CommonLisp$_0$ eine strikte Teilsprache von Common Lisp ist und somit die im Sinne einer Übersetzung möglichen automatischen Sourcecode-Transformationen per se entfallen, da sie syntaktisch und semantisch identischen Code erzeugen müssten. Die Striktheit der Teilsprachenbeziehung hat einen weiteren nutzbringenden Effekt: Die Migration kann inkrementell in einem Lisp-System durchgeführt werden, da jeder Migrationsschritt wieder zu einem gültigen Common-Lisp-Programm führt.

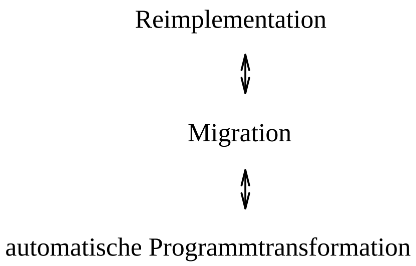

Abbildung 1: Migration

Nachdem der erste Migrationsschritt, die Montage des Programmes (siehe den folgenden Abschnitt) durchgeführt ist, läßt sich der Komplettcompiler als Werkzeug benutzen, um nötige Migrationsschritte aufzudecken und durch eine weitgehende statische Programmanalyse Programmierfehler zu finden.

Es wird in den folgenden Abschnitten deutlich, daß sich das Programm in seiner Struktur durch die Migration nicht wesentlich ändert, sondern höchstens um einige Teile ergänzt wird. Damit kann das migrierte Programm als Ausgangspunkt für die Wartung und Weiterentwicklung dienen, und Migrationsschritte sind nur noch für die dann neu entwickelten Teile nötig. (vgl. Abbildung 2)

2.3 Komplettkompilation

Nachdem ein gültiges CommonLisp$_0$-Programm entstanden ist, kann der Komplettcompiler daraus durch Übersetzung nach C eine eigenständig ablauffähige Applikation generieren. Diese zeichnet sich aus durch

- einen sparsamen Umgang mit Ressourcen, d.h. durch Speicherplatz- und Laufzeiteffizienz,
- Portabilität auf der Basis von C-Sourcecode und
- die Integrationsfähigkeit, zum einen mit Fremdsoftware, zum anderen als Programm oder Modul in Standard-DV-Umgebungen.

Fassen wir die genannten Schritte zusammen, so ergibt sich insgesamt das in Abbildung 2 dargestellte Bild.

3 Methodisches Vorgehen

In diesem Abschnitt soll das methodische Vorgehen bei der Migration von Common-Lisp-Anwendungen nach CommonLisp$_0$ aufgezeigt werden. Zunächst gilt es, festzustellen, ob ein Programm tatsächlich als schlüsselfertige Applikation vorliegt, für die es prinzipiell Sinn macht, sie mit einem explizit ausprogrammierten Hauptprogramm zu versehen und als eigenständig ablauffähiges Programm auszuliefern.

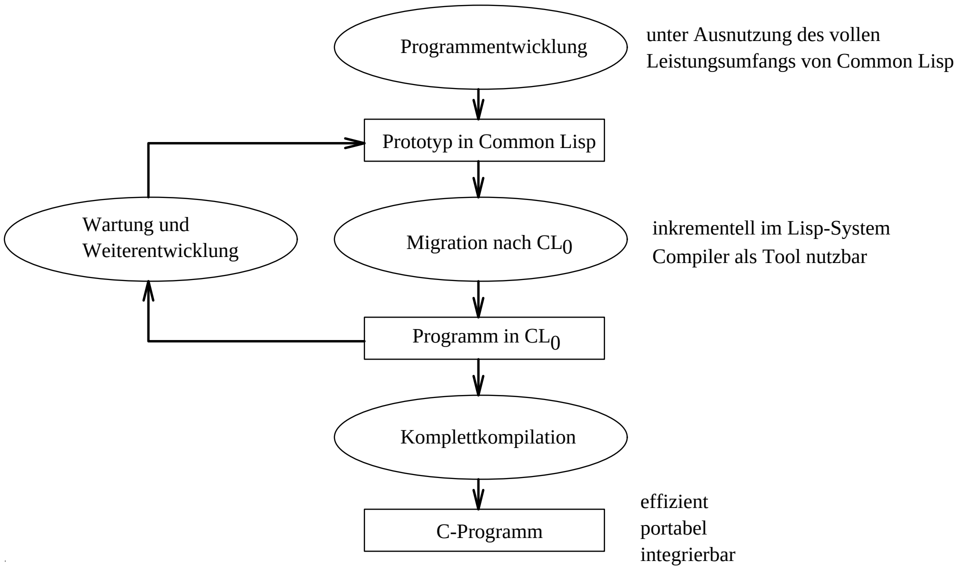

Abbildung 2: Migration und Kompilation

Die nötigen Arbeiten, die wir im folgenden motivieren und genauer beschreiben wollen, lassen sich wie folgt klassifizieren:

1. Montage des Programmes
2. Bereinigung des Programmes
3. Erzeugen eines Hauptprogrammes
4. Elimination von Reflexion
5. Entkopplung vom Entwicklungssystem
6. Modularisierung

Die ersten vier Schritte werden in der Regel in der angegebenen Reihenfolge durchgeführt, die Montage des Programmes ist notwendige Voraussetzung für alle anderen Schritte. Die letzten beiden Schritte sind nur für spezielle Anwendungen erforderlich, insbesondere ist die Modularisierung nur dann zwingend, wenn die separate Kompilation einzelner Module des Programmes angestrebt wird. Dies ist typischerweise bei großen Anwendungen der Fall, oder wenn Fremdsprachenmodule in die Anwendung integriert werden sollen.

3.1 Montage des Programmes

In der Regel sind Lisp-Anwendungen in verschiedenen Dateien abgelegt, und es ist nicht leicht, das eigentliche Programm zwischen ebenfalls vorhandenen Testroutinen und Hilfsmitteln zur Erleichterung des Entwicklungsprozesses auszumachen. Eine mehr oder weniger komplizierte Laderoutine, teilweise mit Unterstützung von *dynamischem Nachladen* (engl. *autoloading*) oder automatischer Rekompilation, Pfadnamenanpassungen u.ä. dient zum Laden des Programmes in ein inkrementelles Lisp-System.

Um die Applikation zu einem Programm im Sinne von CommonLisp_0 zusammenzusetzen, ist folgendes nötig:

- Analyse der Ladereihenfolge beim Laden der Dateien in ein inkrementelles Lisp-System.
- Auffinden der Dateien, die zu der eigentlichen Applikation gehören.
- Erzeugen einer Programmdatei, deren einziger Inhalt Ladeanweisungen auf Hauptprogrammebene für die zum Programm gehörenden Dateien sind.

Der erste Schritt ist auch für die Bereinigung des Programmes erforderlich, um bei eventuellen Mehrfachdefinitionen von Funktionen die gültige, d.h. die *zuletzt* geladene, finden zu können.

Die Haupt-Programmdatei stellt nunmehr das eigentliche Programm dar, das allerdings noch nicht über ein Hauptprogram verfügen muß (vgl. Abschnitt 3.3).

3.2 Bereinigung des Programmes

Da ein inkrementelles Lisp-System Mehrfachdefinitionen von Funktionen zuläßt (die letzte geladene Funktion ist die aktuell verwendete), in CommonLisp_0 Mehrfachdefinitionen aber nicht zulässig ist, sind die nicht aktuellen Funktionsdefinitionen auszukommentieren. Ebenfalls sollten die Programmteile, die ausschließlich zu Testzwecken oder zur Erleichterung des Entwicklungsprozesses dienen, eliminiert werden, soweit dies nicht bereits bei der Montage des Programmes geschehen ist.

Sind nur noch die Bestandteile der eigentlichen Applikation übrig und ist eine Haupt-Programmdatei (vgl. den vorherigen Abschnitt) vorhanden, so kann nunmehr der Compiler als Werkzeug zur statischen Programmanalyse verwendet werden, um weitere Stellen zu finden, an denen Migrationsarbeiten, insbesondere auch zur Bereinigung des Programmes, vorgenommen werden müssen.

Häufig finden sich Aufrufe nicht definierter Funktionen. Diese sind entweder zu definieren oder sie befinden sich in Programmpfaden, die nie durchlaufen werden, so daß sie als *toter Code* gelöscht werden können. Diese Situation kann auftreten, wenn in einem Programm Funktionalität vorgesehen war, die aber nicht oder anders implementiert worden ist.

Funktionalität, die nicht Bestandteil der in [Ste90] angegebenen Sprache Common Lisp ist, sondern von dem speziellen inkrementellen Common-Lisp-System *geborgt* wurde, muß reimplementiert werden.

3.3 Erzeugen eines Hauptprogrammes

Eine Lisp-Anwendung, die nur aus Deklarationen besteht, kann im Zusammenhang mit einem inkrementellen Lisp-System durchaus nutzbar sein, da dieses die Dialog-Umgebung der Anwendung darstellen kann. Zum Zwecke der Komplettkompilation muß allerdings ein *Hauptprogramm* explizit ausprogrammiert werden, das Bestandteil der Haupt-Programmdatei werden sollte.

Dieser Schritt ist in vielen Anwendungen leicht, da nur der Aufruf der Hauptfunktion in die Haupt-Programmdatei zusätzlich aufgenommen werden muß. Existiert allerdings keine Hauptfunktion, so ist in der Regel zusätzliche Design- und Implementationsarbeit zu leisten:

- Der Umfang der dem Programmbenutzer als *Dialogschnittstelle* zur Verfügung gestellten Funktionalität muß festgelegt werden.
- Die so festgelegte Dialogschnittstelle muß implementiert werden. Ihr Aufruf wird zum Hauptprogramm, das nunmehr nur noch antizipierte Dialoge bearbeiten kann. Der Aufruf beliebiger Funktionen der Anwendung wird damit unterbunden, was die Dynamik einschränkt, aber auch die Sicherheit gegen Fehlbedienungen erhöht.

3.4 Elimination von Reflexion

In [Smi82] [Smi84] taucht der Begriff *Reflexion* (engl. *reflection*) zuerst auf. Programme, die „Wissen über ihren eigenen Programmtext" haben, werden dort als reflektiv bezeichnet. Folgerichtig nennen wir Programmiersprachen, die die Programmierung reflektiver Programme gestatten, ebenfalls reflektiv, und die Sprachbestandteile, die das Ausdrücken von Reflexion gestatten, nennen wir reflektive Sprachkonstrukte der Sprache.

M. Wand und D.Friedman [WF86] unterscheiden die Begriffe *reflection* und *reification*. Der Übergang von Programm zu Datum, genauer von einem Abschnitt des abstrakten mathematischen Objektes *Programm* in dessen textuelle Repräsentation, wird dort als *reification* bezeichnet. Als *reflection* wird der eigentlich kritische Übergang von Datum zu Programm bezeichnet, also die Einbindung von Programmtext in das abstrakte Programm, die im ungünstigen Fall eben auch eine Modifikation des Programmes selbst hervorrufen kann.

Lisp, speziell auch Common Lisp, ist eine reflektive Programmiersprache, in der auch der ungünstige Fall der Selbstmodifikation eines Programmes ausdrückbar ist. Um es ganz deutlich zu machen: Es ist nicht der Umgang eines Programmes mit Funktionen oder Prozeduren als Daten erster Klasse (*first class citizens*), der die Reflexion ausmacht. Dieser beinhaltet keine Uminterpretation von Daten in Programm. Es ist vielmehr die in Lisp mögliche freie Uminterpretation von Lisp-Daten, speziell Listen, in Programmausdrücke, also die Ausführung von zum Programm gehörenden, aber im Programmtext i.a. nicht sichtbaren Lisp-Ausdrücken.

Der klassische Vertreter eines reflektiven Sprachbestandteils von Lisp ist die Funktion `(eval ... )` ,die einen als Lisp-Liste übergebenen Ausdruck in der globalen Laufzeitumgebung des Programmes auswertet oder ausführt.

Dem Leser dürfte schnell klar sein, welche Konsequenzen z.B. das Vorhandensein einer Funktionsdefinition der Form

```
(defun f () (eval (read)))
```

in einem Programm hat. Zunächst kann der Übersetzer diese Funktion ohne die Möglichkeit der Kompilation des durch `(read)` eingegebenen Ausdrucks zur Laufzeit nicht implementieren. Zum zweiten können keine globalen Programmoptimierungen mehr durchgeführt werden, die von der vollständigen Kenntnis des Programmtextes abhängen, da dieser eben nicht bekannt ist. Zum dritten erlaubt beispielsweise die Eingabe des Ausdrucks

```
> (defun g (...) ...)
```

die Definition zusätzlicher oder die Modifikation vorhandener Funktionen des Programmes zur Laufzeit, speziell z.B. auch die Modifikation der Funktion `f` selbst.

Insgesamt werden Programme dadurch i.a. unnötig groß, unnötig langsam und aus softwaretechnischer Sicht fehleranfälig und nahezu unwartbar.

Trotzdem ist die durch `(eval... )` und verwandte Sprachkonstrukte gegebene hohe Flexibilität während des Programmentwicklungsprozesses durchaus sehr hilfreich, wie wir in dem abschließenden kleinen Beispiel dieses Abschnittes zeigen wollen.

Reflektive Bestandteile von Common Lisp

Wir beschränken uns im Rahmen dieser Arbeit auf den imperativen Kern der Sprache Common Lisp ohne die objektorientierte Erweiterung CLOS, die mit ihrem Meta-Objekt-Protokoll ebenfalls reflektive Sprachkonstrukte enthält.

Natürlich ist die Funktion `(eval... )` reflektiv. Ebenso klar ist, daß die Funktionen `(compile... )` und `(load... )` die gleiche Mächtigkeit haben. Diese Funktionen dürfen in der Sprache CommonLisp$_0$ nicht vorkommen. Eine Ausnahme bildet `(load... )`, das auf Hauptprogrammebene mit konstantem Dateinamen-Argument erlaubt ist und syntaktisch in den Inhalt der angegebenen Datei expandiert.

Die Funktionen

```
(funcall <function> <arg-1> ... <arg-n>)
(apply <function> <arg-1> ... <arg-k> <arglist>)
```

erlauben in Common Lisp auf der Position des funktionalen Arguments auch Symbole (interpretiert als Funktionsnamen) und Listen der Form `(lambda <parlist> <body>)` (als Funktionen interpretiert). Sie erlauben auf der Argumentposition Ausdrücke, die zu Funktionsnamen oder Lambda-Listen evaluieren. Damit läßt sich die Funktion `eval` natürlich leicht durch

```
(defun eval (form) (funcall (list 'LAMBDA () form)))
```

programmieren. Für diese Funktionen wie auch für `(mapcar... )`, `(mapc... )` usw. ist in CommonLisp$_0$ der Definitionsbereich eingeschränkt, indem nur Funktionen, nicht aber Funktionsnamen oder Lambda-Listen auf der Position des funktionalen Arguments zugelassen sind.

Die Funktion `(symbol-function... )` erlaubt in Common Lisp durch

```
(setf (symbol-function 'f) ...)
```

die Redefinition globaler Funktionen. Diese Funktion ist in CommonLisp$_0$ nicht erlaubt.

Es gibt einige weitere Stellen, z.B. in Typdefinitionen, beim Einlesen konstanter Strukturen oder in der Funktion `(format... )`, die in CommonLisp$_0$ eingeschränkt sind, da sich auch hier die Möglichkeit der Interpretation eines beliebigen Ausdrucks zur Laufzeit des Programmes programmieren läßt. Im Rahmen der vorliegenden Arbeit soll auf eine vollständige und genaue Darstellung verzichtet werden.

Programmspezifische Interpretation

Wie bereits aus den Bemerkungen des vorherigen Abschnitts deutlich geworden ist, verbietet CommonLisp$_0$ alle reflektiven Sprachbestandteile von Common Lisp, teiweise durch Streichen, teilweise dadurch, daß der Definitionsbereich von Funktionen derart eingeschränkt ist, daß ein dynamischer Übergang von Datum zu Programm nicht mehr möglich ist. Das Ziel der Migration bezüglich reflektiver Programmbestandteile ist also deren Eliminierung.

Die universelle Lisp-Funktion `(eval... )` zur Interpretation beliebiger Lisp-Ausdrücke stellt häufig sehr viel mehr Funktionalität zur Verfügung, als mit ihrer Verwendung intendiert ist. Ein Grund dafür ist die komplizierte Semantik von `(eval... )`. In der Regel weiß der Programmierer jedoch sehr genau, welche Art von Ausdrücken durch die Verwendung von `(eval... )` interpretiert werden sollen. In den meisten Fällen handelt es sich um den Aufruf von Funktionen, die ihrerseits durch das Programm definiert werden. Hier liegt es nahe, statt des allgemeinen Interpretationsmechanismus' einen programmspezifischen Interpreter auszuprogrammieren, der genau die gewünschten Ausdrücke interpretieren kann und alle weiteren, die beispielsweise durch Programmierfehler oder Fehlbedienungen entstehen, explizit abweist. Wir wollen dies in einem Beispiel tun, indem wir

```
(eval (read))
```

unter Ausnutzung der zusätzlichen Information, daß nur Ausdrücke der Form

```
e ::= <number> | (+ e1 ... en) | (- e1 ... en) | (* e1 ... en)
```

interpretiert werden sollen in den Ausdruck

```
(my-eval (read))
```

ändern, wobei die Funktion `my-eval` defiert ist durch

```
(defun my-eval (form)
  (cond ((numberp form) form)
        ((eq (car form) '+)
           (apply #'+ (mapcar #'my-eval (cdr form))))
        ((eq (car form) '-)
           (apply #'- (mapcar #'my-eval (cdr form))))
        ((eq (car form) '*)
           (apply #'* (mapcar #'my-eval (cdr form))))
        (T (error "unknown form"))))
```

Es ist klar, daß es sich hierbei um eine Programmtransformation handelt, die nicht semantisch äquivalent ist. Das geänderte Programm verhält sich aber bzgl. der intendierten „äußeren“ Semantik äquivalent und ist zudem sicherer und wartbarer geworden, da Fehlbedienungen ausgeschlossen wurden und mehr Informationen explizit niedergeschrieben sind.

Natürlich lassen sich bestimmte Anwendungen von `(eval...)`, `(funcall...)` oder `(apply...)` auch ohne Kenntnis der äußeren Semantik des Programmes semantisch äquivalent transformieren, beispielsweise gilt

```
(eval '(f arg-1 ... arg-n))         ==  (f arg-1 ... arg-n)
(eval (cons 'f arglist))            ==  (apply #'f arglist)
(apply 'f arglist)                  ==  (apply #'f arglist)
(apply '(lambda (...)...) arglist)
       ==  (apply #'(lambda (...)...) arglist)
```

Diese Fälle kann natürlich auch der Übersetzer selbst abdecken, so daß sie für die Migration eher uninteressant sind. Es wird an dieser Stelle noch einmal deutlich, daß die Migration ihrem Wesen nach eine Aufgabe ist, die der Programmierer eigenhändig und unter Ausnutzung seiner Kenntnis über die Bedeutung oder die intendierte äußere Semantik seines Programmes vornehmen sollte.

Ersetzen von Ausdrücken durch Closures

Sehr häufig finden sich in Lisp-Programmen globale konstante Datenstrukturen, die der Speicherung von zu interpretierenden Ausdrücken oder von in bestimmten Fällen aufzurufenden Funktionen dienen. Als Beispiel seien hier eine Assoziationsliste mit Zuordnungen von Symbolen zu Interpretationsfunktionen

```
(defvar *fun-alist* '((+ . eval-plus) (- . eval-minus) ...))
```

oder auch einfach eine Liste von Ausdrücken

```
(defvar *expr-list* '((+ 3 *x*) (* *y* *x*) ...))
```

genannt. Typischerweise tauchen dann in dem Programm Ausdrücke der Form

```
(apply (cdr (assoc '+ *fun-alist*)) args)
```

oder

```
(eval (second *expr-list*))
```

auf, die natürlich nicht im Sprachumfang von CommonLisp_0 enthalten sind, da z.B. der Ausdruck `(cdr (assoc '+ *fun-a-list*))` ein Symbol, nicht aber eine Funktion liefern wird.

In derartigen Fällen ermöglicht der Übergang von Ausdrücken zu *closures* bzw. der Übergang von Symbolen zu den entsprechenden Funktionen selbst den nötigen Migrationsschritt. Im ersten Fall ersetzen wir die konstante Assoziationsliste durch eine, die statt der Symbole `eval-plus` usw. die Funktionen `#'eval-plus` usw. enthält:

```
(defvar *fun-alist*
  (list (cons '+ #'eval-plus)
        (cons '- #'eval-minus)
        ...))
```

und können den Aufruf sogar unverändert lassen. Im zweiten Fall ersetzen wir die Liste von Ausdrücken durch eine Liste von *closures*

```
(defvar *expr-list*
  (list #'(lanbda () (* 3 *x*))
        #'(lambda () (* *y* *x*))
        ...))
```

und ersetzen den Ausdruck `(eval (second *expr-list*))` durch den Funktionsaufruf

```
(funcall (second *expr-list*))
```

3.5 Ein Beispiel

An einem sehr kleinen Beispiel, einem Programm zum symbolischen Defferenzieren von Polynomen, wollen wir eine typische Programmentwicklung, die entstehende Lisp-Anwendung und die wesentlichen Schritte bei der Migration zu einem CommonLisp_0-Programm aufzeigen.

Programmentwicklung

Die Aufgabe ist die Entwicklung eines symbolischen Differenzierers für Polynome:

Eingabe: $p(x) = a_n x^n + \cdots + a_1 x + a_0$

Ausgabe: $p'(x) = n a_n x^{n-1} + \cdots + a_1$

Wir gehen im Sinne des *rapid prototyping* vor und vereinfachen die Aufgabenstellung in einer für Lisp typischen Weise, indem wir die Programmierung der konkreten Text-Ein/Ausgabe verschieben und zunächst den Kern, nämlich die Differenzierung einer abstrakten Listendarstellung des Polynoms, entwickeln. Die zu Testzwecken nötige Eingabe stellt der Lisp-Reader zur Verfügung. Damit ist die Aufgabe nun sehr einfach geworden:

Wir stellen das Polynom in abstrakter Syntax dar, aufgebaut aus Ausdrücken der Form

$$\begin{array}{c}\texttt{(PLUS } e_1 \ e_2\texttt{)} \\ \texttt{(MINUS } e_1 \ e_2\texttt{)} \\ \texttt{(MULT } e_1 \ e_2\texttt{)} \\ \texttt{(EXPP } e \ n\texttt{) .}\end{array}$$

Das Polynom $p(x) = x^2 + 2x + 2$ beispielsweise wird dargestellt durch die folgende Lisp-Liste:

```
(PLUS (EXPP x 2) (PLUS (MULT 2 x) 2))
```

Mit den Ableitungsregeln

- $c' = y' = 0$ für Konstanten c und Variablen $y \not\equiv x$
- $x' = 1$
- $(f + g)' = f' + g'$
- $(f\,g)' = f'g + fg'$
- $(f^n)' = nf'f^{n-1}$ für $n \geq 2$

ist es nun einfach, die Funktion `diff` auszuprogrammieren:

```
(defun diff (e x)
  (cond ((numberp e) 0)
        ((eq e x) 1)
        ((symbolp e) 0)
        ((consp e)
          (case (first e)
            ((PLUS MINUS) ... )
            ((MULT) (list 'PLUS
                      (list 'MULT (diff (second e) x)
                            (third e))
                      (list 'MULT (second e)
                            (diff (third e) x))))
            ((EXPP) ... )
            (t e)))
        (t e)))
```

Die komplette Funktionsdefinition findet sich im Anhang. Ein erster Test ergibt nun, daß wir mit dem Ergebnis noch nicht zufrieden sind, da das differenzierte Polynom noch erheblich vereinfacht werden kann:

$$p(x) = x^2 + 2$$

```
(diff '(PLUS (EXPP x 2) 2) 'x) =
(PLUS (MULT 1 (MULT 2 (EXPP x (MINUS 2 1)))) 0)
```

Die Vereinfachung kann durch partielles Auswerten des Ergebnisses geschehen, indem wir die Funktionen `PLUS`, `MINUS`, `MULT`, `EXPP` schreiben und den entstandenen Ausdruck evaluieren. Als Beispiel seien die Funktionen `MULT` und `EXPP` mit den entsprechenden Auswertungsregeln angegeben:

$$(\texttt{MULT}\ e_1\ e_2) = \begin{cases} e_1 e_2 & \text{, falls } e_1, e_2 \text{ Zahlen sind} \\ 0 & \text{, falls } e_1 = 0 \text{ oder } e_2 = 0 \\ e_2 & \text{, falls } e_1 = 1 \\ e_1 & \text{, falls } e_2 = 1 \\ (\texttt{MULT}\ e_1\ e_2) & \text{, sonst} \end{cases}$$

```
(defun MULT (e1 e2)
  (cond ((and (numberp e1) (numberp e2)) (* e1 e2))
        ((or (eql 0 e1) (eql 0 e2)) 0)
        ((eql e1 1) e2)
        ((eql e2 1) e1)
        (t (list 'MULT e1 e2))))
```

$$(\texttt{EXPP}\ e\ n) = \begin{cases} e^n & \text{, falls } e, n \text{ Zahlen sind} \\ 1 & \text{, falls } n = 0 \\ e & \text{, falls } n = 1 \\ (\texttt{EXPP}\ e\ n) & \text{, sonst} \end{cases}$$

```
(defun EXPP (e n)
  (cond ((and (numberp e) (numberp n)) (^ e n))
        ((eql 0 n) 1)
        ((eql n 1) e)
        (t (list 'EXPP e n))))
```

Die Funktion zur partiellen Auswertung der entstehenden Ausdrücke, die wir `simplify` nennen wollen, kann dann einfach die Lisp-Funktion `eval` aufrufen. Damit ergibt sich eine erste Version des Programmes, vervollständigt durch Definitionen der Funktionen `parse` und `unparse`, die aufgrund der verabredeten Vereinfachung nur die als Argument übergebene abstrakte Darstellung des Polynoms identisch zurückgeben:

```
(defun diff ... )

(defun PLUS ... )
(defun MINUS ... )
(defun MULT ... )
(defun EXPP ... )
```

```
(defun simplify (p) (eval p))
(defun parse (p) p)
(defun unparse (p) p)

(defun test (p)
  (unparse (simplify (diff (parse p) 'x))))
```

Das komplette Programm findet sich im Anhang.

Migration

Zur Montage des Programmes ist nichts zu tun. Eine zusätzliche Haupt-Programmdatei ist nicht erforderlich. Wir können also den Übersetzer auf das Programm ansetzen, um Stellen zu finden, bei denen Migrationsschritte nötig sind. Die statisch semantische Analyse des Übersetzers zeigt uns, daß weder die Funktion `(eval... )`, deren Aufruf in der Funktion `simplify` auftritt, noch die Funktion `(^... )`, die in `EXPP` aufgerufen wird, im Sprachumfang von CommonLisp_0 vorhanden sind.

Die Funktion `(^... )` ist tatsächlich keine definierte Funktion in Common Lisp. Wir haben einen Programmierfehler entdeckt, der deshalb bislang nicht zutage getreten ist, weil der entsprechende Fall in der Funktion `EXPP` nie erreicht wurde und auch nie erreicht werden wird. Ausdrücke der Form `(EXPP` n_1 n_2`)` mit Zahlen n_1 und n_2 tauchen in den Ergebnissen von `diff` nicht auf, wenn wir sicherstellen, daß die Funktion `parse` reduzierte abstrakte Repräsentationen liefert. Damit kann der Aufruf von `(^... )` beispielsweise durch die Fehlermeldung

```
(error "Fatal error in function EXPP")
```

ersetzt werden.

Um den Aufruf der Funktion `(eval... )` zu eliminieren, entscheiden wir uns für die Programmierung eines programmspezifischen Interpretierers `diff-eval`, den wir stattdessen in der Funktion `simplify` aufrufen. Dieser läßt sich nach demselben Schema programmieren, das wir bereits im Abschnitt 3.4 benutzt haben:

```
(defun diff-eval (e)
  (cond
    ((or (numberp e) (symbolp e)) e)
    ((eq (first e) 'PLUS)
      (apply #'PLUS (mapcar #'diff-eval (rest e))))
    ((eq (first e) 'MINUS)
      (apply #'MINUS (mapcar #'diff-eval (rest e))))
    ((eq (first e) 'MULT)
      (apply #'MULT (mapcar #'diff-eval (rest e))))
    ((eq (first e) 'EXPP)
      (apply #'EXPP (mapcar #'diff-eval (rest e))))
    (t (error "Unknown expression"))))
```

Ergänzen wir das entstandene Programm nun noch um ein Hauptprogramm, das aus einer Schleife besteht, die jeweils ein zu differenzierendes Polynom einliest und das differenzierte Polynom ausdruckt, dann entsteht letztlich als Ergebnis der Migration das folgende Programm, das im Anhang vollständig wiedergegeben ist:

```
(defun diff ... )

(defun PLUS ... )
(defun MINUS ... )
(defun MULT ... )

(defun EXPP (e n)
  (cond ((and (numberp e) (numberp n)) (error "Fatal error in expp"))
        ((eql 0 n) 1)
        ((eql n 1) e)
        (t (list 'EXPP e n))))

(defun diff-eval (e) ... )
(defun simplify (p) (diff-eval p))

(defun parse (p) p)
(defun unparse (p) p)

(defun test (p)
  (unparse (simplify (diff (parse p) 'x))))

(defun main ()
  (loop
    (princ "Enter expression : ")
    (let ((e (read)))
      (if (member e '(quit exit end stop halt))
        (return)
        (format t "~&~s~%" (test1 e))))))

(main)
```

Dieses Programm können wir nun komplett übersetzen. Die generierte eigenständig ablauffähige Applikation führt das Hauptprogramm, also den Aufruf der Funktion `(main)` aus. Im Anschluß würden die Funktionen `parse` und `unparse` entwickelt werden, die eine Textrepräsentation eines Polynoms lesen und in abstrakte Syntax umwandeln bzw. erzeugen.

4 Resultate

Mit dem in dieser Arbeit vorgestellten methodischen Vorgehen sind im Rahmen des APPLY-Projektes eine Reihe mittelgroßer bis großer Lisp-Applikationen auch externer

Partner des Projektes erfolgreich migriert, übersetzt und auch auf kleine Anwendungsrechner portiert worden. Exemplarisch seien hier zwei Expertensystemanwendungen und ein Interpreter für eine KI-Sprache genannt, die im folgenden mit XPS1, XPS2 und KI-S bezeichnet werden sollen.

Die folgende Tabelle zeigt für diese Anwendungen einen Vergleich der Codegrößen. Dabei sind die Sourcecode-Größen, die Binärcode-Größe eines mit herkömmlicher Technik erzeugten Lisp-Images in Allegro-CL 4.1 und die Binärcode-Größen der mit der hier vorgestellten Technik übersetzten ausführbaren Programme zueinander in Beziehung gesetzt:

	XPS 1	XPS 2	KI-S
Lisp-Source-Code	10120 lines of code 485 KB	10671 lines of code 505 KB	3185 lines of code 92.4 KB
C-Source-Code	2073.5 KB	2212.1 KB	556 KB
Lisp-Image	8.6 MB	8.8 MB	7.8 MB
Programm (SPARC)	794.6 KB	942.0 KB	464.0 KB
Programm (AT-386)	620.9 KB	747.9 KB	178.0 KB

Alle hier angegebenen Programme sind auf PC-AT's ohne Einschränkungen ablauffähig, sogar auf Notebooks mit nur 2MByte Hauptspeicher.

Fassen wir die Vorteile des Vorgehens im Hinblick auf die entstandenen ablauffähigen Programme zusammen, so ergibt sich:

1. Sparsamer Umgang mit Ressourcen
   - Drastische Verringerung der Code-Größen
   - Ablauffähigkeit auf Kleinrechnern (Expertensystem auf einem 2MB Notebook
2. Portabilität
   - maschinenunabhängiger C-Code
3. Integrierbarkeit
   - als Programme in Standard-DV-Umgebungen
   - mit C-basierter Standardsoftware, z.B.
     - Bedienoberflächen
     - Datenbanken
     - Kommunikationssoftware

Das in dieser Arbeit vorgeschlagene Vorgehen, um Lisp-Anwendungen zu entwickeln, zu migrieren und sie schließlich als eigenständig ablauffähige Applikationen zu implementieren, hat bei einer Reihe von nichttrivialen Anwendungen zum Erfolg geführt.

Dabei bleibt die Produktivität beim Programmieren in Lisp erhalten, wegweisende Entwicklungen in der symbolischen Datenverarbeitung und Wissensverarbeitung werden nicht gehemmt und das schnelle Entstehen von Prototypen wird begünstigt.

Wir haben ein methodisches Vorgehen vorgeschlagen und ausgearbeitet, wie die entstandenen Prototypen zu Anwendungen weiterentwickelt werden können, die den klassischen

Anforderungen an Laufzeit- und Speicherplatzeffizienz genügen, und die insbesondere der Anforderung gerecht werden, sich in vorhandene Datenverarbeitungsumwelten integrieren zu lassen.

Sie können Industriestandards nutzen, wie beispielsweise Datenbanken, Bedienoberflächen oder Kommunikationssoftware. Dies ist nicht neu. Auch die bekannten kommerziell verfügbaren Common-Lisp-Systeme bieten durch ein Foreign-Language-Interface die technischen Voraussetzungen dazu.

Neu ist, daß sich Lisp-Programme selbst diesen Standards entsprechend verhalten, sich also als Programme oder Module wie solche jeder anderen Programmiersprache nutzen lassen. Anwender müssen nicht Lisp lernen, um Lisp-Programme nutzen zu können, und die Anwendungsumgebung muß nur noch die Rechenleistung für die Anwendung vorhalten, nicht aber die für die Nutzung eines kompletten Lisp-Systems nötige.

Der Preis, den man für diese Vorteile zu zahlen hat, ist der Arbeitsaufwand für die Migration. Wir meinen, daß dieser aus softwaretechnischer Sicht sowieso nötig ist, um von Prototypen zu Anwendungen zu gelangen.

5 Vergleichbare Arbeiten

Es gibt heute eine Reihe weiterer Übersetzerbauprojekte, in denen C erfolgreich als Zielsprache der Übersetzung höherer Sprachen verwendet wird. Dazu gehören Übersetzer von Fortran, Scheme und ML nach C. Aber auch die Übersetzung von Common Lisp nach C erfährt mehr und mehr Beachtung. Es seien hier z.B. KCL (Kyoto Common Lisp), WCL (Wade Common Lisp) und auch der kommerziell verfügbare Chestnut Lisp-to-C-Translator genannt.

KCL [TM86] ist ein in C implementiertes inkrementelles Lisp-System. Der inkrementelle Übersetzer erzeugt zwar C-Code, der aber nicht eigenständig ablauffähig ist, sondern durch dynamisches Linken als inkrementell übersetzter Programmbestandteil im Entwicklungssystem nutzbar wird.

WCL [Hen92] verfolgt einen eher mit dem von APPLY vergleichbaren Ansatz. In WCL werden Common-Lisp-Programme ebenfalls nach C übersetzt, die ablauffähigen Programme benötigen allerdings die vollständige Common-Lisp-Laufzeitbibliothek in Form einer sogenannten *shared library*. Damit ist die Portabilität der Anwendungen im wesentlichen auf Unix-Umgebungen beschränkt, zudem besteht wegen der Reflektivität des vollen Common Lisp kaum eine nennenswerte Möglichkeit, globale Programmoptimierungen auszunutzen.

Am ehesten vergleichbar mit dem APPLY-Ansatz ist der kommerziell verfügbare Chestnut Lisp-to-C-Translator, der eigenständig ablauffähigen C-Code erzeugt, dies aber mit einem Hauptaugenmerk auf die Lesbarkeit und Wartbarkeit des entstehenden C-Codes, der im APPLY-Ansatz nur als portabler Zwischencode dient und im Hinblick auf Effizienz, nicht aber in erster Linie im Hinblick auf Lesbarkeit erzeugt wird.

Literatur

[BCF+91] Harry Bretthauer, Thomas Christaller, Horst Friedrich, Wolfgang Goerigk, Winfried Heicking, Dieter Hovekamp, Heinz Knutzen, Jürgen Kopp, E. Ulrich Kriegel, Ingo Mohr, Rainer Rosenmüller und Friedemann Simon. Spezifikation der Funktionalität des APPLY-Systems. APPLY-Arbeitspapier APPLY/IfKI/I.1/1, CAU/GMD/IfKI/VW-GEDAS, Sankt Augustin, Oktober 1991.

[BGK91] O. Burkart, W. Goerigk und H. Knutzen. CLICC: A New Approach to the Compilation from Common Lisp Programs to C. In *Workshop der Fachgruppe 2.1.4. "Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, erschienen als Bericht Nr. 8/91-I des Instituts für angewandte Mathematik und Informatik der Univ. Münster, 1991.

[GS91] W. Goerigk und F. Simon. Zielsetzungen im Verbundvorhaben APPLY. In *Workshop der Fachgruppe 2.1.4. "Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, erschienen als Bericht Nr. 8/91-I des Instituts für angewandte Mathematik und Informatik der Univ. Münster, 1991.

[Hen92] W. Hennessey. WCL: Delivering Efficient Common Lisp Applications under Unix. In *Proc. of the 1992 ACM Conference on Lisp and Functional Programming*, S. 260–269, San Francisco, CA, 1992.

[KG93] U. Knemeyer und J.-M. Graf von der Schulenburg. „Expertensysteme" - Welche Faktoren fördern und hemmen die Implementation und Diffusion der Technologie in der Versicherungswirtschaft. In F. Puppe und A. Günter, Hrsg., *Expertensysteme '93*, Berlin, Heidelberg, New York, 1993. Informatik aktuell, Springer Verlag.

[Knu92] Heinz Knutzen. CommonLisp$_0$-Sprachdefinition. APPLY-Arbeitspapier APPLY/CAU/II.2/1, Christian-Albrechts-Universität, Kiel, April 1992.

[MBG90] P. Mertens, V. Barowski und W. Geis. *Betriebliche Expertensystem-Anwendungen.* 2. Auflage, Springer Verlag, Berlin, Heidelberg, New York, 1990.

[Smi82] B.C. Smith. Reflection and Semantics in a Procedural Language. Technischer Bericht MIT-LCS-TR-272, Massachusetts Institute of Technology, Cambridge, MA, 1982.

[Smi84] B.C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages POPL'84*, S. 23–35, 1984.

[Ste84] G.L. Steele. *Common Lisp: The Language.* Digital Press, Bedford, MA, 1984.

[Ste90] G.L. Steele. *Common Lisp: The Language. Second Edition.* Digital Press, Bedford, MA, 1990.

[TM86] Y. Taiichi und H. Masami. Kyoto Common Lisp Report. Technical report, Research Institute for Mathematical Science, Kyoto University, 1986.

[Weu93] H. Weule. Expertensysteme im industriellen Einsatz. In F. Puppe und A. Günter, Hrsg., *Expertensysteme '93*, Berlin, Heidelberg, New York, 1993. Informatik aktuell, Springer Verlag.

[WF86] M. Wand und D.P. Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *Proceedings of the 1986 ACM-Conference on Lisp and Functional Programming*, S. 298–307, Cambridge, MA, 1986.

A Das Beispielprogramm

```
;;;-----------------------------------------------------------------
;;;
;;; Symbolisches Differenzieren von Polynomen
;;;
;;;-----------------------------------------------------------------

(defun diff (e x)
  (cond ((numberp e) 0)
        ((eq e x) 1)
        ((symbolp e) 0)
        ((consp e)
         (case (first e)
           ((PLUS MINUS) (list (first e)
                               (diff (second e) x)
                               (diff (third e) x)))
           ((MULT) (list 'PLUS
                         (list 'MULT (diff (second e) x) (third e))
                         (list 'MULT (second e) (diff (third e) x))))
           ((EXPP) (list 'MULT
                        (diff (second e) x)
                        (list 'MULT
                              (third e)
                              (list 'EXPP
                                    (second e)
                                    (list 'MINUS (third e) 1)))))
           (t e)))
        (t e)))

;;;-----------------------------------------------------------------

(defun PLUS (e1 e2)
   ;;;
   ;;;  (PLUS n1 n2) = n1 + n2
   ;;;  (PLUS 0 e2) = e2
   ;;;  (PLUS e1 0) = e1
   ;;;
  (cond ((and (numberp e1) (numberp e2)) (+ e1 e2))
        ((eql 0 e1) e2)
        ((eql 0 e2) e1)
        (t (list 'PLUS e1 e2))))

(defun MINUS (e1 e2)
   ;;;
   ;;;  (MINUS n1 n2) = n1 - n2
```

```
   ;;;  (MINUS e1 0) = e1
   ;;;
  (cond ((and (numberp e1) (numberp e2)) (- e1 e2))
        ((eql 0 e2) e1)
        (t (list 'MINUS e1 e2))))

(defun MULT (e1 e2)
   ;;;
   ;;;  (MULT n1 n2) = n1 * n2
   ;;;  (MULT 0 e2) = (MULT e1 0) = 0
   ;;;  (MULT 1 e2) = e2
   ;;;  (MULT e1 1) = e1
   ;;;
  (cond ((and (numberp e1) (numberp e2)) (* e1 e2))
        ((or (eql 0 e1) (eql 0 e2)) 0)
        ((eql e1 1) e2)
        ((eql e2 1) e1)
        (t (list 'MULT e1 e2))))

(defun EXPP (e n)
   ;;;
   ;;;  (EXPP n1 n2) = n1 ^ n2
   ;;;  (EXPP e 0) = 1
   ;;;  (EXPP e 1) = e
   ;;;
  (cond ((and (numberp e) (numberp n)) (^ e n))
        ((eql 0 n) 1)
        ((eql n 1) e)
        (t (list 'EXPP e n))))

;;;---------------------------------------------------------------------

(defun parse (p) p)
(defun unparse (p) p)

(defun simplify (p) (eval p))

;;;---------------------------------------------------------------------

(defun test (p)
  (setq x 'x)
  (unparse (simplify (diff (parse p) 'x))) )

;;;---------------------------------------------------------------------
```

B Das Beispielprogramm nach der Migration

```
;;;-----------------------------------------------------------------
;;;
;;; Symbolisches Differenzieren von Polynomen (nach der Migration)
;;;
;;;-----------------------------------------------------------------

(defun diff (e x)
  (cond ((numberp e) 0)
        ((eq e x) 1)
        ((symbolp e) 0)
        ((consp e)
         (case (first e)
           ((PLUS MINUS) (list (first e)
                               (diff (second e) x)
                               (diff (third e) x)))
           ((MULT) (list 'PLUS
                         (list 'MULT (diff (second e) x) (third e))
                         (list 'MULT (second e) (diff (third e) x))))
           ((EXPP) (list 'MULT
                        (diff (second e) x)
                        (list 'MULT
                              (third e)
                              (list 'EXPP
                                    (second e)
                                    (list 'MINUS (third e) 1)))))
           (t e)))
        (t e)))

;;;-----------------------------------------------------------------

(defun PLUS (e1 e2)
   ;;;
   ;;;  (PLUS n1 n2) = n1 + n2
   ;;;  (PLUS 0 e2) = e2
   ;;;  (PLUS e1 0) = e1
   ;;;
  (cond ((and (numberp e1) (numberp e2)) (+ e1 e2))
        ((eql 0 e1) e2)
        ((eql 0 e2) e1)
        (t (list 'PLUS e1 e2))))

(defun MINUS (e1 e2)
   ;;;
   ;;;  (MINUS n1 n2) = n1 - n2
```

```
   ;;;  (MINUS e1 0) = e1
   ;;;
  (cond ((and (numberp e1) (numberp e2)) (- e1 e2))
        ((eql 0 e2) e1)
        (t (list 'MINUS e1 e2))))

(defun MULT (e1 e2)
   ;;;
   ;;;  (MULT n1 n2) = n1 * n2
   ;;;  (MULT 0 e2) = (MULT e1 0) = 0
   ;;;  (MULT 1 e2) = e2
   ;;;  (MULT e1 1) = e1
   ;;;
  (cond ((and (numberp e1) (numberp e2)) (* e1 e2))
        ((or (eql 0 e1) (eql 0 e2)) 0)
        ((eql e1 1) e2)
        ((eql e2 1) e1)
        (t (list 'MULT e1 e2))))


(defun EXPP (e n)
   ;;;
   ;;;  (EXPP n1 n2) = n1 ^ n2
   ;;;  (EXPP e 0) = 1
   ;;;  (EXPP e 1) = e
   ;;;
  (cond ((and (numberp e) (numberp n)) (error "Fatal error in expp"))
        ((eql 0 n) 1)
        ((eql n 1) e)
        (t (list 'EXPP e n))))

;;;---------------------------------------------------------------------

(defun parse (p) p)
(defun unparse (p) p)

(defun simplify (p) (diff-eval p))

;;;---------------------------------------------------------------------

(defun test (p)
  (unparse (simplify (diff (parse p) 'x))) )

;;;---------------------------------------------------------------------

(defun diff-eval (e)
```

```
  (cond ((or (numberp e) (symbolp e)) e)
        ((eq (first e) 'PLUS)
          (apply #'PLUS (mapcar #'diff-eval (rest e))))
        ((eq (first e) 'MINUS)
          (apply #'MINUS (mapcar #'diff-eval (rest e))))
        ((eq (first e) 'MULT)
          (apply #'MULT (mapcar #'diff-eval (rest e))))
        ((eq (first e) 'EXPP)
          (apply #'EXPP (mapcar #'diff-eval (rest e))))
        (t (error "Unknown expression; can't evaluate ~s." e))))

;;;---------------------------------------------------------------------

(defun main ()
  (loop
    (princ "Enter expression : ")
    (let ((e (read)))
      (if (member e '(quit exit end stop halt))
        (return)
        (format t "~&~s~%" (test e))))))

(main)
```