Verbesserte prozedurale Programmiersprachen

Zusammenfassung der wichtigsten Arbeitsergebnisse anstelle einer Habilitationsschrift

Dr. Christian Heinlein
Fakultät für Ingenieurwissenschaften und Informatik
Universität Ulm

Oktober 2006

Kurzfassung. Trotz ihrer weiten Verbreitung und allgemein hohen Akzeptanz, leiden objektorientierte Programmiersprachen (und damit auch ihre Benutzer) an einigen unangenehmen Problemen, für die es trotz intensiver Forschungsarbeit über viele Jahre noch keine rundum zufriedenstellenden Lösungen gibt. Unter anderem sind dies das *expression problem*, das *binary method problem*, das *diamond inheritance problem* und Probleme mit sog. *crosscutting concerns*. Da viele dieser Probleme in klassischen *prozeduralen Programmiersprachen* überhaupt nicht existieren, weil sie mit Vererbung und dynamischem Binden zusammenhängen, stellt sich die berechtigte Frage, ob sie nicht einfach durch eine Rückkehr zu diesen Sprachen überwunden werden können, sofern die genannten Konzepte anderweitig zur Verfügung gestellt werden können. Und tatsächlich erhält man durch relativ einfache Verallgemeinerungen der beiden Kernkonzepte prozeduraler Sprachen – Datenstrukturen und Prozeduren – *verbesserte prozedurale Programmiersprachen*, die objektorientierten Sprachen bezüglich Ausdrucksmächtigkeit und Flexibilität nicht nur ebenbürtig, sondern sogar überlegen sind. Unter anderem lassen sich für bestehende Typen nachträglich Obertypen definieren, virtuelle und nicht-virtuelle Vererbung können problemlos miteinander kombiniert werden, und Objekte können ihren Typ zur Laufzeit ändern, um beispielsweise eine Entwicklung von einer Person zu einem Studenten und wieder zurück zu „durchleben". Trotz dieser Flexibilität, die ansonsten bestenfalls von dynamisch typisierten Sprachen angeboten wird, sind verbesserte prozedurale Sprachen *statisch typsicher*.
Die angesprochene Verallgemeinerung von Prozeduren deckt neben *single*, *multiple* und *predicate-based method dispatch* auch *before*, *after* und *around advice* aspektorientierter Sprachen ab, ohne hierfür zusätzliche spezialisierte Sprachmittel einführen zu müssen. Damit sind verbesserte prozedurale Sprachen insbesondere geeignet, bestehende Software-Systeme nachträglich *nicht-invasiv* (d. h. ohne bestehenden Code zu modifizieren oder neu übersetzen zu müssen) in ihrer Funktionalität zu erweitern oder zu modifizieren und damit dem Problem der *unanticipated software evolution* zu begegnen.

1 Einleitung

1.1 Hintergrund und Motivation

Trotz ihrer weiten Verbreitung und allgemein hohen Akzeptanz, leiden *objektorientierte Programmiersprachen* (mit statischer Typprüfung) an einigen unangenehmen Problemen, für die es trotz intensiver Forschungsarbeit über viele Jahre hinweg noch keine rundum zufriedenstellenden Lösungen gibt. Unter anderem sind dies Probleme im Zusammenhang mit *diamond inheritance* (unter welchen Umständen soll eine indirekt mehrfach geerbte Klasse in der abgeleiteten Klasse repliziert werden, sofern die zugrundeliegende Sprache überhaupt Mehrfachvererbung von Klassen anbietet; vgl. z. B. [28, 34, 23, 24, 25] für eine ausführlichere Darstellung der Problematik sowie unterschiedliche Lösungsansätze), Probleme durch *binary methods* (die kovariante Anpassung eines Parametertyps in einer Unterklasse würde zu Lücken in der statischen Typsicherheit führen; vgl. [14] für eine ausführliche Darstellung der Problematik) und fehlende Unterstützung für *dynamic object evolution* (ein einmal erzeugtes Objekt kann seinen Typ zur Laufzeit nicht mehr verändern, um z. B. eine Entwicklung von einer Person zu einem Studenten und wieder zurück zu durchleben). Außerdem gibt es nach wie vor keine einfache und überzeugende Lösung des *Erweiterbarkeitsproblems* (*expression problem*; um nachträglich neue Methoden zu einer Klassenhierarchie hinzuzufügen, muss der Quellcode der Klassen verändert und neu übersetzt werden, d. h. operationale Erweiterungen eines bestehenden Systems sind nicht *modular* möglich; vgl. Abschnitt 2), und auch sog. *crosscutting concerns* (Querschnittsbelange wie z. B. Synchronisation oder Protokollierung, die nichts mit der Kernfunktionalität eines Systems zu tun haben; vgl. z. B. [27, 35]) widersetzen sich einer modularen Implementierung.

Anders als mit objektorientierten Sprachen, ist es mit *aspektorientierten Programmiersprachen* zumindest möglich, die beiden letztgenannten Probleme zu lösen. Allerdings geht die Zunahme an Ausdrucksmächtigkeit und Flexibilität bei der Entwicklung imperativer Programmiersprachen[1], ausgehend von klassischen prozeduralen Sprachen über modulare und objektorientierte Sprachen hin zu aspektorientierten Sprachen, auch mit einer erheblichen Zunahme an Komplexität der jeweiligen Sprachen einher (vgl. Abb. 1, die exemplarisch einige bekannte Vertreter der verschiedenen Kategorien zeigt): während die Sprachberichte zu Pascal [38] oder Modula [39] beispielsweise noch auf jeweils 30 Seiten einer Zeitschrift abgedruckt werden konnten, füllen die Spezifikationen von Java [21] oder C++ [34] dicke Bücher, und aspektorientierte Sprachen wie AspectJ [27] oder AspectC++ [33] erweitern diese Sprachen nochmals um zahlreiche neue Konzepte.

Um dieser unerwünschten Zunahme an Komplexität zu begegnen, wurde in der vorliegenden Arbeit ein vollkommen anderer Weg beschritten, um die eingangs erwähnten Probleme zu lösen: Anstatt objektorientierte Sprachen noch weiter auszubauen, wurden klassische prozedurale bzw. modulare Programmiersprachen als Ausgangspunkt verwendet, deren Kernkonzepte *Datenstrukturen* und *Prozeduren* geeignet erweitert wurden, um sog. *offene Typen* und *globale virtuelle Funktionen* zu erhalten. Zusammen mit dem z. B. von Modula-2 [40] und Oberon [41] bekannten *Modulkonzept*, konnten so *verbesserte prozedurale Programmiersprachen* entwickelt werden, die objektorientierten Sprachen bezüglich Ausdrucksmächtigkeit und Flexibilität nicht nur ebenbürtig, sondern sogar überlegen sind und dennoch mit wesentlich weniger Sprachmitteln und Konzepten auskommen. Darüber hinaus erhält man einige wesentliche Vorteile aspektorientierter Sprachen, ohne hierfür zusätzliche spezialisierte Sprachmittel einführen zu müssen.

1.2 Ansatz

Verbesserte prozedurale Programmiersprachen bestehen aus drei *orthogonalen Kernkonzepten* (offene Typen, globale virtuelle Funktionen und Module) sowie einigen *zusätzlichen Konzepten* (z. B. Nullwerte, Sequenzen und benutzerdefinierte Operatorsymbole), die unabhängig von den Kernkonzepten

[1] Unter dem Begriff *imperative Programmiersprachen* werden hier alle Sprachen subsumiert, die auf dem Grundprinzip einer von-Neumann-Rechnerarchitektur basieren, im Gegensatz beispielsweise zu funktionalen Sprachen, die auf dem Grundprinzip seiteneffektfreier Funktionen basieren.

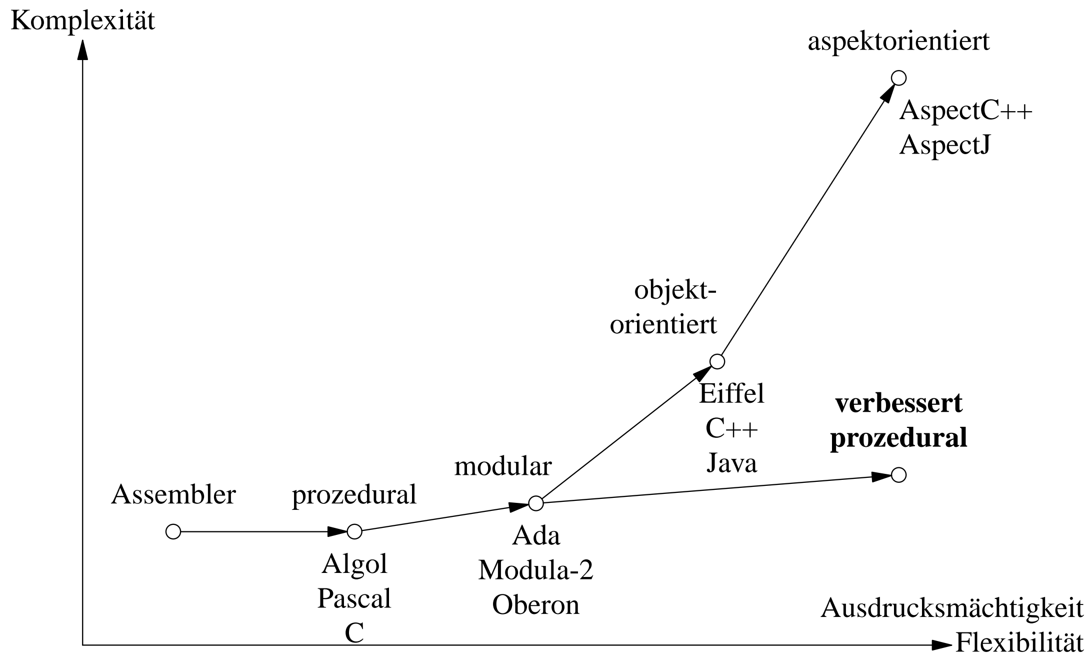

Abbildung 1: Flexibilität und Komplexität imperativer Programmiersprachen

nützlich, aber nicht unbedingt erforderlich sind. Im Gegensatz zu objektorientierten Sprachen, in denen Datenstrukturen, Operationen und Module fest zu Klassen zusammengeschnürt sind (vgl. Abb. 2), können diese drei Grundbausteine in verbesserten prozeduralen Sprachen beliebig miteinander kombiniert werden. Unter anderem können mehrere offene Typen in einem einzigen Modul zusammengefasst werden (beispielsweise ein Containertyp mit einem zugehörigen Iteratortyp), damit die zugehörigen Funktionen wechselseitig auf ihre Attribute zugreifen können, ohne hierfür komplexe Sprachmittel wie z. B. geschachtelte Klassen in Java [21] oder „Freundschaftsbeziehungen“ in C++ [34] einführen zu müssen.

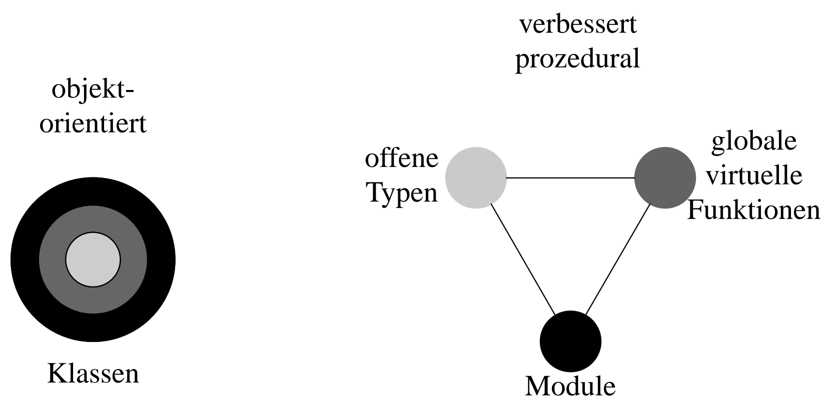

Abbildung 2: Datenstrukturen (●), Operationen (●) und Module (●)

Alle Konzepte verbesserter prozeduraler Sprachen sind an sich *sprachunabhängig* und könnten daher in unterschiedliche imperative Programmiersprachen eingebettet werden. Exemplarisch wurden sie als Spracherweiterungen für C++ (und teilweise auch für andere Sprachen) realisiert, wodurch jedoch zahlreiche bestehende Elemente dieser Sprache überflüssig werden, insbesondere so komplexe

Konzepte wie Klassen, Vererbung und virtuelle Elementfunktionen sowie fehlerträchtige Elemente wie Zeiger und manuelle Speicherverwaltung. Daher ist die resultierende Sprache C+++ konzeptuell tatsächlich wesentlich „schlanker“ als ihre Basissprache C++.

1.3 Gliederung

Im nachfolgenden Abschnitt 2 wird das Erweiterbarkeitsproblem als eines der fundamentalen Probleme objektorientierter Programmiersprachen genauer vorgestellt sowie die bekanntesten Lösungsvorschläge kritisch analysiert. Anschließend werden in Abschnitt 3 die drei Kernkonzepte verbesserter prozeduraler Sprachen vorgestellt und durch Beispiele illustriert. Für weiterführende Informationen wird auf die in Abschnitt 6 genannten Veröffentlichungen verwiesen, in denen die Konzepte wesentlich ausführlicher vorgestellt und mit verwandten Arbeiten verglichen werden. In Abschnitt 4 wird genauer auf die praktische Umsetzung der Konzepte und ihre Verwendung in Lehrveranstaltungen eingegangen.

Abschnitt 5 fasst die wesentlichen Ergebnisse noch einmal zusammen und gibt einen Ausblick auf zukünftige Arbeiten. Abschnitt 6 enthält eine kommentierte Zusammenstellung der relevanten Veröffentlichungen, die anstelle einer Habilitationsschrift vorgelegt werden, während Abschnitt 7 Referenzen auf verwendete Literatur enthält.

2 Das Erweiterbarkeitsproblem

2.1 Problembeschreibung

Abbildung 3 illustriert das Erweiterbarkeitsproblem anhand des üblicherweise hierfür verwendeten Beispiels arithmetischer Ausdrücke (*arithmetic expressions*, daher u. a. auch der Name *expression problem* [36]): Um ein Software-System zu implementieren, das unterschiedliche Operationen (wie z. B. `eval` und `print`) auf unterschiedlichen Varianten von Ausdrücken (wie z. B. `Const` für konstante Ausdrücke, `Add` für Additionen usw.) anbietet, kann man den dadurch aufgespannten zweidimensionalen Raum entweder horizontal oder vertikal partitionieren.

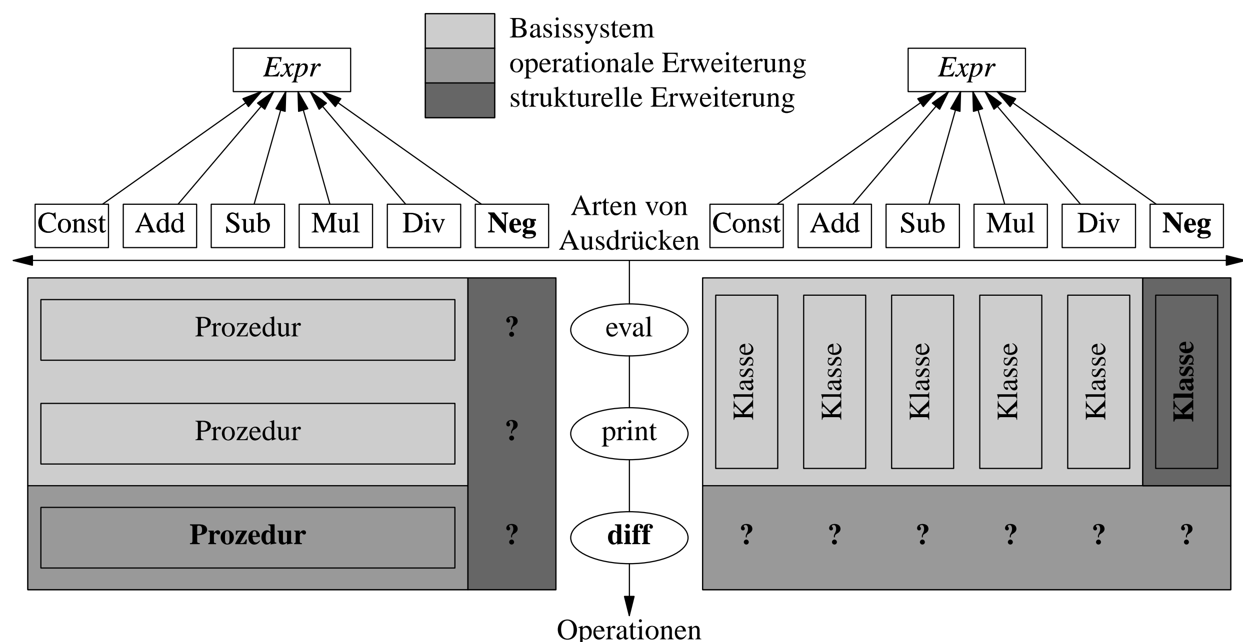

Abbildung 3: Illustration des Erweiterbarkeitsproblems

Im einen Fall implementiert man jede Operation als *Prozedur* in einer prozeduralen Sprache (oder auch als Funktion in einer funktionalen Sprache), die anhand einer Fallunterscheidung (bzw. *pattern matching*) die unterschiedlichen Arten von Ausdrücken behandelt (linke Seite der Abbildung). Bei dieser Vorgehensweise können zusätzliche Operationen (wie z. B. `diff` für symbolisches Differenzieren von Ausdrücken) problemlos zu einem existierenden System hinzugefügt werden, ohne hierfür bereits vorhandenen Code ändern oder neu übersetzen zu müssen, d. h. *operationale Erweiterungen* eines bestehenden Systems sind in *modularer* Weise möglich. Andererseits erfordert eine nachträgliche Definition zusätzlicher Datenstrukturen (z. B. `Neg` für Negationen) eine Modifikation aller bestehenden Operationen, um die zusätzlichen Fälle zu berücksichtigen, d. h. eine *strukturelle Erweiterung* eines bestehenden Systems ist nicht modular möglich.

Im anderen Fall implementiert man jede Variante von Ausdrücken als *Klasse* in einer objektorientierten Sprache, die für jede Operation eine (dynamisch gebundene) Methode enthält (rechte Seite der Abbildung). Bei dieser Vorgehensweise können zusätzliche Datenstrukturen in Form zusätzlicher (Unter-)Klassen problemlos zu einem existierenden System hinzugefügt werden, ohne die bereits vorhandenen Klassen ändern oder neu übersetzen zu müssen, d. h. hier sind strukturelle Erweiterungen modular möglich. Andererseits erfordert eine nachträgliche Definition zusätzlicher Operationen eine Modifikation aller bestehenden Klassen, um die zusätzlichen Methoden zu ergänzen, d. h. hier sind operationale Erweiterungen nicht modular möglich.

Bei beiden Vorgehensweisen wird daher nur *eine* von zwei Dimensionen modularer Erweiterbarkeit unterstützt. Eine dritte Dimension, die nachträgliche Erweiterung oder Modifikation des *Verhaltens* bestehender Operationen (wie z. B. die nachträgliche Synchronisation von `print`-Operationen) wird in keinem Fall unterstützt.

2.2 Bekannte (objektorientierte) Lösungsvorschläge

Da das Erweiterbarkeitsproblem seit langem bekannt und für praktische Anwendungen hochgradig relevant ist, gibt es im objektorientierten Umfeld zahlreiche Lösungsvorschläge, um eine bestehende Klassenhierarchie nachträglich um zusätzliche Operationen zu erweitern, die jedoch alle mehr oder weniger große Schwachstellen aufweisen.

Die am häufigsten vorgeschlagene „Lösung“ ist die Verwendung von Unterklassen, in denen die zusätzlichen Operationen einfach als neue Methoden definiert werden. Das Hauptproblem dabei ist, dass bereits existierender Code Objekte der ursprünglichen Klassen erzeugt, die die neuen Operationen folglich nicht besitzen. (Dies könnte – bei entsprechender Vorausplanung – durch den Einsatz eines Fabrikmusters [19] vermieden werden.) Ebenso erwarten bereits existierende Methoden syntaktisch Objekte der ursprünglichen Klassen. Aufgrund der üblichen Subtyp-Polymorphie sind Objekte der neuen Klassen zwar kompatibel zu den ursprünglichen Klassen, aber bei entsprechenden Zuweisungen oder Parameterübergaben geht syntaktisch die Typinformation verloren, dass es sich um Objekte der neuen Klassen handelt. Daher muss diese immer dort, wo die zusätzlichen Operationen benötigt werden, durch dynamische Typtests und bedingte Downcasts wieder verfügbar gemacht werden. Schließlich müssen die neuen Klassen nicht nur von den entsprechenden alten abgeleitet werden (z. B. `Const2` von `Const`), sondern – um polymorphe Verwendbarkeit zu ermöglichen – auch von einer neuen Unterklasse `Expr2` der Wurzelklasse `Expr`, in der die neuen Methoden abstrakt deklariert werden. Somit erhält man eine relativ komplexe Klassenhierarchie mit Mehrfachvererbung, was von vielen objektorientierten Sprachen gar nicht unterstützt wird.

Ein weiterer häufig vorgeschlagener Lösungsweg ist der Einsatz des Besuchermusters (*visitor pattern* [19]). Ein wesentlicher Nachteil dabei ist, dass die Verwendung dieses Musters von Anfang an geplant werden muss, d. h. dass sich damit keine *unerwarteten* Erweiterungen realisieren lassen. Außerdem ist die resultierende Klassenstruktur ebenfalls relativ komplex. Und schließlich erlauben Lösungen mit Besuchermuster keine nachträglichen Strukturerweiterungen, d. h. man befindet sich letztlich in der gleichen Situation wie mit prozeduralen Sprachen, in der lediglich operationale Erweiterungen modular möglich sind.

Neben diesen weitverbreiteten Lösungsvorschlägen gibt es verschiedene Ansätze, das Erweiterbarkeitsproblem mit Hilfe generischer Typen zu lösen [36]. Auch hier ist wieder eine sorgfältige Voraus-

planung erforderlich, und die Komplexität des resultierenden Codes übersteigt jedes für normale Anwendungen akzeptable Maß, so dass diese Lösungen bestenfalls von theoretischem Wert sind. Außerdem ist einzuwenden, dass hier generische Typen zur Lösung eines Problems verwendet werden, das von Natur aus überhaupt nicht generisch ist.

Da sich das Erweiterbarkeitsproblem also einer einfachen Lösung mit den üblichen Ausdrucksmitteln objektorientierter Sprachen hartnäckig widersetzt, versuchen andere Ansätze, das Problem durch geeignete Spracherweiterungen in den Griff zu bekommen. Hier sind vor allem Sprachen wie CLOS [26], Dylan [17] und MultiJava [16, 30] zu nennen, die die Definition von *Multimethoden* außerhalb von Klassen erlauben. Tatsächlich sind globale virtuelle Funktionen, mit denen das Erweiterbarkeitsproblem in verbesserten prozeduralen Sprachen gelöst wird, konzeptuell ähnlich, aber noch flexibler als Multimethoden.

Wie bereits erwähnt, bieten auch aspektorientierte Sprachen zusätzliche Sprachmittel an (konkret z. B. *inter-type member declarations* in AspectJ [27] und *introductions* in AspectC++ [33]), mit denen das Erweiterbarkeitsproblem zufriedenstellend gelöst werden kann. Allerdings führt dieser Ansatz zu einer künstlichen und unnötigen syntaktischen Unterscheidung zwischen den ursprünglichen Operationen eines Systems (die als gewöhnliche Methoden in Klassen definiert werden können) und den nachträglich hinzugefügten Operationen (die als *inter-type member declarations* bzw. *introductions* in Aspekten definiert werden müssen), während globale virtuelle Funktionen eine einheitliche syntaktische Form für beide Arten von Operationen bieten. Dadurch wird u. a. zum Ausdruck gebracht, dass nachträgliche Erweiterungen eines bestehenden Systems nicht als seltener Spezialfall, sondern eher als Normalfall betrachtet werden.

2.3 Modulare Erweiterung von Datenstrukturen

Obwohl das Erweiterbarkeitsproblem meist nur für Operationen formuliert wird und sich die typisch objektorientierten Lösungsvorschläge auf das nachträgliche Hinzufügen von Operationen zu bestehenden Klassen beschränken, sollte erwähnt werden, dass das Problem in gleicher Weise auch für die modulare Erweiterung von Datenstrukturen besteht. Abgesehen von den bereits erwähnten *inter-type member declarations* bzw. *introductions* aspektorientierter Sprachen, mit denen nicht nur Methoden, sondern auch Datenfelder zu bestehenden Klassen hinzugefügt werden können, existieren hierfür jedoch keine bekannten Lösungsvorschläge. Analog zu Abschnitt 2.2 erhält man auch hier wieder eine künstliche Unterscheidung zwischen ursprünglichen und nachträglich hinzugefügten Datenfeldern, während die in dieser Arbeit vorgestellten offenen Typen beide Kategorien gleich behandeln, d. h. nachträglich hinzugefügte Datenfelder nicht als Ausnahme, sondern als Normalfall betrachten.

3 Kernkonzepte verbesserter prozeduraler Programmiersprachen

3.1 Offene Typen

3.1.1 Konzept

Die Grundidee offener Typen [1] besteht darin, Typ- und Attributdeklarationen voneinander zu *trennen*, damit Attribute *inkrementell* zu einem bereits bestehenden Typ hinzugefügt werden können, ohne dessen ursprüngliche Definition verändern zu müssen. Die so definierten Attribute eines Typs sind grundsätzlich *optional*, d. h. ein Objekt eines Typs muss nicht notwendigerweise Werte für alle Attribute des Typs besitzen, und die zugrundeliegende Implementierung sorgt dafür, dass für jedes Objekt nur die tatsächlich benötigten Attribute gespeichert werden und Platz beanspruchen (vgl. Abschnitt 4.2). Nichtsdestotrotz kann für jedes Objekt auf jedes statisch definierte Attribut seines Typs zugegriffen werden, wobei man ggf. einen wohldefinierten *Nullwert* erhält, der die Abwesenheit eines echten Attributwerts anzeigt. Objekte offener Typen besitzen *Referenzsemantik* ähnlich wie Objekte

in Java (d. h. Variablen offener Typen enthalten lediglich Objektreferenzen), und nicht mehr benötigte Objekte werden automatisch freigegeben (*garbage collection*).

3.1.2 Typen und Attribute

Um einen offenen Typ `Person` mit einem *einwertigen Attribut* `name` vom Typ `string` sowie einem *mehrwertigen Attribut* `gnames` (given names) ebenfalls vom Typ `string` zu definieren (vgl. Abb. 4), wird der Typ zunächst mit Hilfe des Schlüsselworts `typename` deklariert und anschließend die Attribute hinzugefügt:

```
typename Person;                 // Offener Typ Person.
Person -> string name;           // Einwertiges Attribut name.
Person ->> string gnames;        // Mehrwertiges Attribut gnames.
```

Anschließend können mit Hilfe des vordefinierten *Attribut-Initialisierungs-Konstruktors*, der mit einer Reihe von Attribut/Wert-Paaren aufgerufen wird, konkrete Objekte dieses Typs erzeugt werden, zum Beispiel (vgl. Abb. 5):

```
// Person-Objekte erzeugen.
Person p = Person(@gnames, "Christian")(@name, "Heinlein");
Person q = Person(@gnames, "Charles")(@gnames, "Anthony")
  (@gnames, "Richard")(@name, "Hoare");
```

Wenn häufig gleichartige Objekte erzeugt werden sollen, ist es ratsam, einen *benutzerdefinierten Konstruktor* zu definieren, der einer globalen Funktion entspricht, die einerseits den Namen des Typs besitzt und andererseits ein Objekt dieses Typs erzeugt und zurückliefert, typischerweise durch einen Aufruf des vordefinierten Konstruktors:

```
// Benutzerdefinierter Konstruktor.
Person (string g, string n) {
    // Person mit Vorname g und Name n erzeugen.
    return Person(@gnames, g)(@name, n);
}

// Aufruf des benutzerdefinierten Konstruktors.
Person p = Person("Christian", "Heinlein");
```

Abbildung 4: Offener Typ `Person` mit zwei Attributen `name` und `gnames`

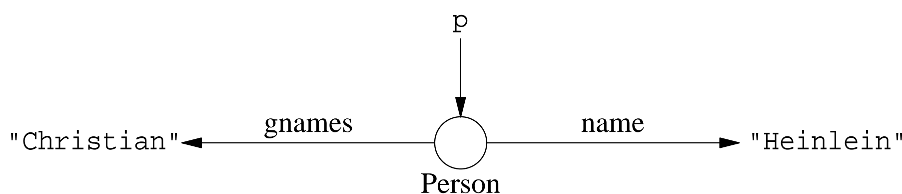

Abbildung 5: Objekt des offenen Typs `Person` mit zwei Attributwerten

Zum lesenden Zugriff auf die Attributwerte eines Objekts wird der `@`-Operator verwendet, der ähnlich wie der Punkt-Operator in anderen Sprachen angewandt wird, z. B. `p@name`, um den Wert des Attributs `name` des Objekts `p` (genauer des von `p` referenzierten Objekts) zu erhalten. Wenn auf diese Weise auf ein mehrwertiges Attribut wie z. B. `q@gnames` zugegriffen wird, erhält man alle für dieses Attribut gespeicherten Werte in Form einer *Sequenz*, auf deren Elemente mit dem bekannten Indexoperator zugegriffen werden kann, z. B. `q@gnames[2]`, um den zweiten Vornamen der Person `q` (im konkreten Beispiel „Anthony") zu erhalten. Ebenso ist es möglich, mit einer speziellen Schleife alle Werte einer solchen Sequenz zu durchlaufen.

Wie bereits in Abschnitt 3.1.1 erwähnt, erhält man beim Zugriff auf einen nicht vorhandenen Attributwert eines Objekts einen wohldefinierten Nullwert des entsprechenden Zieltyps. Insbesondere erhält man eine *Nullreferenz*, d. h. eine Referenz auf *kein Objekt* bzw. ein virtuelles *Nullobjekt*, wenn der Zieltyp des Attributs selbst wieder ein offener Typ ist. Greift man auf die Attributwerte dieses gedachten Nullobjekts zu, so erhält man ebenfalls wieder entsprechende Nullwerte.

3.1.3 Modulare Erweiterbarkeit

Das besondere an offenen Typen im Gegensatz zu klassischen Recordtypen und Klassen ist die Möglichkeit, sie nachträglich in modularer Weise um zusätzliche Attribute erweitern zu können. Beispielsweise kann der oben definierte Typ `Person` jederzeit um ein weiteres Attribut `ssno` (social security number) vom Typ `integer` erweitert werden, ohne dass hierfür die ursprüngliche Typdefinition verändert werden müsste:

```
Person -> integer ssno;
```

Prinzipiell kann diese Erweiterung sogar in einem separaten Modul erfolgen, für dessen Übersetzung nicht einmal der Quellcode des ursprünglichen Moduls benötigt wird. Greift man mittels `p@ssno` auf den Wert dieses neuen Attributs zu, erhält man für das oben erzeugte `Person`-Objekt `p` einen Nullwert, da das Objekt noch keinen Wert für dieses Attribut besitzt. Dies kann z. B. wie folgt in einer Fallunterscheidung abgefragt werden:

```
if (p@ssno) ...... // Attributwert vorhanden.
else ......        // Attributwert nicht vorhanden.
```

3.1.4 Statische Typsicherheit

Trotz dieser zusätzlichen Flexibilität sind offene Typen *statisch typsicher*. Dies liegt zum einen daran, dass bei jeder Attributdeklaration Basis- und Zieltyp des Attributs festgelegt werden, und dass bei einem Attributzugriff `obj@attr` der Attributname `attr` statisch bekannt sein muss und den statischen Typ des Objekts `obj` als Basistyp besitzen muss. Für nachträglich definierte Attribute bedeutet dies insbesondere, dass sie erst im Anschluss an ihre Deklaration verwendet werden können und grundsätzlich nur in dem Modul bekannt sind, in dem sie definiert wurden. Da jedes Modul außerdem einen unabhängigen Namensbereich darstellt, führen versehentlich oder absichtlich gleich benannte Attribute eines Typs in unterschiedlichen Modulen nicht zu Namenskonflikten, sondern stellen vielmehr voneinander unabhängige Dinge dar. Schließlich sorgt die Tatsache, dass beim Zugriff auf einen nicht vorhandenen Attributwert ein wohldefinierter Nullwert geliefert wird, dafür, dass niemals versehentlich auf undefinierte Werte oder Speicherbereiche zugegriffen werden kann.

3.1.5 Bidirektionale Relationen

Beziehungen zwischen zwei offenen Typen, die in beiden Richtungen verwendbar sein sollen, können prinzipiell als Paare von Attributen modelliert werden, zum Beispiel:

```
Person ->> Car cars;     // Autos einer Person.
Car -> Person owner;     // Besitzer eines Autos.
```

Diese Attribute definieren zusammen eine 1:N-Beziehung zwischen `Person` und `Car`, die in der ei-

nen Richtung `cars` und in der anderen Richtung `owner` heißt. Allerdings muss man bei dieser Vorgehensweise als Programmierer selbst darauf achten, die beiden Richtungen wechselseitig konsistent zu halten. Beispielsweise muss zu einer Anweisung wie `p(@cars, c)`, die `c` zu den `cars` von `p` hinzufügt, auch die duale Anweisung `c(@owner, p)` ausgeführt werden, die `p` als `owner` von `c` vereinbart. Falls `c` bereits einen Vorbesitzer `q` hat, muss es darüber hinaus aus dessen Sequenz von `cars` entfernt werden. Fasst man die beiden Attribute jedoch zu einer einzigen *bidirektionalen Relation* zusammen:

```
Person owner <->> Car cars;
```

so kümmert sich das Laufzeitsystem automatisch um alle notwendigen wechselseitigen Aktualisierungen von Attributwerten, d. h. wenn man jetzt *eine* der Anweisungen `p(@cars, c)` *oder* `c(@owner, p)` ausführt, wird die andere automatisch mitausgeführt, ebenso wie die eventuell erforderliche Löschoperation, die `c` aus der Sequenz `cars` seines Vorbesitzers entfernt.

3.1.6 Anonyme und automatische Attribute und Relationen

Wenn für ein einwertiges Attribut kein Name angegeben wird, besitzt es implizit den Namen seines Zieltyps, zum Beispiel:

```
typename Address;       // Typ Address mit geeigneten Attributen.
......
Person -> Address;      // Anonymes Attribut von Person mit Typ Address.

p(@Address, q@Address); // Attributwert abfragen und setzen.
```

Ebenso ist es möglich, einen oder beide Namen einer bidirektionalen Relation wegzulassen.

Wenn dem Pfeil in einer Attributdeklaration ein Ausrufezeichen folgt, wird das Attribut bei Bedarf *automatisch* angewandt, um ein Objekt des Ursprungstyps (links vom Pfeil) in ein Objekt bzw. einen Wert des Zieltyps (rechts vom Pfeil) umzuwandeln, zum Beispiel:

```
Car ->! string name;    // Automatisches Attribut name.
Car c = Car(@name, ...);// Objekt des Typs Car.

string s = c;           // Implizite Anwendung des Attributs.
                        // Entspricht: string s = c@name;
```

Entsprechend kann dem Pfeil in einer Relationsdeklaration ein Ausrufezeichen folgen oder vorangehen, um anzuzeigen, dass die Relation in der entsprechenden Richtung bei Bedarf automatisch angewandt werden kann. Häufig sind automatische Attribute und Relationen auch anonym.

3.1.7 Vererbung und Subtyp-Polymorphie

Automatische (und typischerweise anonyme) 1:1-Relationen können dazu verwendet werden, einen offenen Typ als „Untertyp" eines anderen zu definieren und damit die objektorientierten Konzepte von Vererbung und Subtyp-Polymorphie nachzubilden, ohne hierfür zusätzliche spezialisierte Sprachmittel einführen zu müssen. Um beispielsweise `Student` als Untertyp von `Person` zu definieren, werden beide Typen zunächst unabhängig voneinander definiert, und anschließend wird durch eine automatische 1:1-Relation die „Vererbungsbeziehung" vereinbart:

```
// Person mit Name und Vornamen.
typename Person;
Person -> string name;
Person ->> string gnames;
```

```
// Student mit Matrikelnummer.
typename Student;
Student -> integer number;

// Vererbungsbeziehung.
Student <->! Person;
```

Ebenso werden zur Erzeugung eines „Studenten“ zunächst zwei unabhängige Objekte der Typen `Person` und `Student` erzeugt, die anschließend über die 1:1-Relation miteinander verbunden werden (vgl. Abb. 6):

```
// Student mit Vorname g, Name n und Matrikelnummer m erzeugen.
Student (string g, string n, integer m) {
    Person p = Person(@gnames, g)(@name, n);
    Student s = Student(@number, m);
    s(@Person, p); // impliziert: p(@Student, s);
    return s;
}
```

Die Tatsache, dass ein einzelner „Student“ implementierungstechnisch aus zwei assoziierten Teilobjekten besteht, bleibt bei der anschließenden Verwendung des „Objekts“ jedoch vollständig verborgen. Insbesondere können die von `Person` „geerbten“ Attribute direkt auf ein `Student`-Objekt `s` angewandt werden (z. B. `s@name`), da dieses mit Hilfe der automatischen Relation (die in Abb. 6 durch einen Pfeil mit offener Spitze dargestellt ist) bei Bedarf automatisch in das assoziierte `Person`-Objekt `s@Person` umgewandelt wird (so dass `s@name` vom Compiler automatisch durch `s@Person@name` ersetzt wird). Aus demselben Grund ist es möglich, `Student`-Objekte polymorph als `Person`-Objekte zu verwenden.

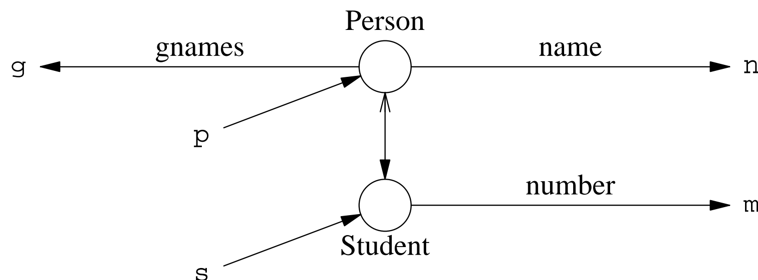

Abbildung 6: `Student`-Objekt mit assoziiertem `Person`-Teilobjekt

Da die Relation zwischen `Student` und `Person` bidirektional ist, kann sie auch in der umgekehrten Richtung verwendet werden, um herauszufinden, ob es zu einem `Person`-Objekt ein assoziiertes `Student`-Objekt gibt, und ggf. auf dieses zuzugreifen, d. h. quasi einen *dynamischen Typtest* und einen *bedingten Downcast* auszuführen:

```
Student s = Student(g, n, m);    // Student erzeugen ...
Person p = s;                    // ... und polymorph als Person verwenden.
                                 // Entspricht: Person p = s@Person;

if (p@Student) {                 // Dynamischer Typtest ...
    cout << p@Student@number;    // ... und bedingter Downcast.
}
```

```
// Entspricht in Java:
if (p instanceof Student) {                 // Typtest.
    System.out.print(((Student)p).number); // Downcast.
}
```

Auch hierfür sind, im Gegensatz zu objektorientierten Sprachen, keinerlei spezialisierte Sprachmittel erforderlich. Tatsächlich werden durch automatische Attribute und Relationen die ansonsten getrennten Konzepte von *Aggregation*, *Vererbung* und *impliziten Typumwandlungen* zu einem einzigen Konzept zusammengefasst.

3.1.8 Dynamische Objektevolution

Die Tatsache, dass ein Objekt eines „Untertyps" in Wirklichkeit durch ein Netzwerk assoziierter Teilobjekte repräsentiert wird, dessen Struktur bei Bedarf auch verändert werden kann, führt unmittelbar zum Konzept der *dynamischen Objektevolution*, d. h. zu der Möglichkeit, den „dynamischen Typ" eines Objekts nach seiner Erzeugung verändern zu können. Um beispielsweise eine gewöhnliche Person nachträglich in einen Studenten zu transformieren, genügt es, ein assoziiertes `Student`-Objekt zu erzeugen und mit dem bereits vorhandenen `Person`-Objekt zu verbinden (vgl. wiederum Abb. 6):

```
Person p = Person(g, n); // Person-Objekt.
if (p@Student) ......    // Dynamischer Typtest liefert false.

Student(@Person, p)(@number, m); // Transformation in einen Studenten.
if (p@Student) ......    // Dynamischer Typtest liefert jetzt true.
```

Eine solche Transformation von einem allgemeinen zu einem spezielleren Typ ist offensichtlich statisch typsicher, da alle für den allgemeinen Typ definierten Operationen (wie z. B. Zugriff auf die Attribute `name` und `gnames`) auch für den spezielleren Typ definiert sind.

Um einen Studenten zurück in eine gewöhnliche Person zu transformieren, kann sein `Student`-Teilobjekt *explizit gelöscht* werden, was zur Folge hat, dass alle eventuell noch bestehenden Referenzen auf dieses Objekt automatisch null werden:

```
Student s = p@Student;  // Downcast.

~ p@Student;            // Rücktransformation in eine Person
                        // durch Löschen des Student-Teilobjekts.

if (p@Student) ......   // Dyn. Typtest liefert jetzt wieder false.
if (s) ......           // s wurde automatisch null.
```

Tatsächlich ist auch eine solche Rücktransformation von einem speziellen zu einem allgemeineren Typ statisch typsicher, da Operationen, die nur für den speziellen Typ definiert sind (wie z. B. Zugriff auf das Attribut `number`), nur über Referenzen mit entsprechendem *statischen Typ* ausgeführt werden können und diese durch die Rücktransformation automatisch null werden, d. h. insbesondere wohldefiniert bleiben. Somit zeigen Operationen auf solchen Referenzen (wie z. B. `s@number`) dasselbe Verhalten wie Operationen auf Nullreferenzen im allgemeinen. (Im konkreten Beispiel erhält man einen Nullwert, vgl. Abschnitt 3.1.2.)

3.1.9 Weiterführende Information

Neben der in Abschnitt 3.1.7 gezeigten einfachen Vererbung, kann mit automatischen 1:1-Relationen auch mehrfache und wiederholte Vererbung modelliert werden (z. B. `EmployedStudent`, d. h. eine Person, die sowohl Angestellter als auch Student ist, oder `DoubleDegreeStudent`, d. h. eine Person, die für zwei Studiengänge eingeschrieben ist), wiederum ohne zusätzliche Sprachmittel zur Unterscheidung zwischen einfach und mehrfach geerbten Obertypen (*diamond* vs. *repeated inheritance*)

einführen zu müssen. Außerdem ist es problemlos möglich, nachträglich Obertypen für bereits definierte Typen einzuführen.

Veröffentlichung [1] enthält eine sehr umfassende Darstellung des Konzepts offener Typen mit zahlreichen Anwendungsbeispielen, u. a. zu *diamond* und *repeated inheritance* und zu *dynamic object evolution*. Außerdem werden die wesentlichen Ideen zur effizienten Implementierung offener Typen vorgestellt und das Konzept mit zahlreichen verwandten Ansätzen verglichen, u. a.:

- *inter-type member declarations* [27] bzw. *introductions* [33] in aspektorientierten Sprachen;
- *subject-oriented programming* [22];
- *multi-dimensional separation of concerns* [35];
- dynamisch typisierte Sprachen und Systeme wie z. B. *Smalltalk* [20];
- prototypbasierte Sprachen wie z. B. *Self* [37];
- Modellierungssprachen wie z. B. *ER-Diagramme* [15] und *UML* [31];
- terminologische Logiken [13, 29].

Kurz zusammengefasst, unterscheiden sich offene Typen von allen vergleichbaren *statisch typisierten* Ansätzen dadurch, dass sie absolut *modular* erweiterbar sind, d. h. dass zusätzliche Attribute nicht nur unabhängig vom ursprünglichen Quellcode definiert werden können, sondern dass auch weder beim Übersetzen noch beim Zusammenlinken noch beim Ausführen von Code irgendeine Form von *weaving* erforderlich ist. Außerdem ist es der einzige bekannte Ansatz, der statische Typsicherheit mit dynamischer Objektevolution und der Möglichkeit dynamischer Erweiterungen von Datenstrukturen zur Laufzeit kombiniert (vgl. Abschnitt 3.3.3), die ansonsten bestenfalls von *dynamisch typisierten* Ansätzen angeboten werden.

Abschnitt 3.5 des Vorlesungsskripts [11] bietet ebenfalls eine gut verständliche Einführung in die wesentlichen Konzepte offener Typen.

3.2 Globale virtuelle Funktionen

3.2.1 Konzept

Globale virtuelle Funktionen [3, 5] stellen eine Kombination *globaler Funktionen* und *virtueller Elementfunktionen* (d. h. dynamisch gebundener Methoden) dar, d. h. es handelt sich um globale Funktionen, die bei Bedarf *redefiniert* werden können. Eine Redefinition ersetzt hierbei die ursprüngliche Definition vollständig, d. h. alle Aufrufe der Funktion werden automatisch zur Redefinition umgeleitet, die die ursprüngliche Definition bei Bedarf jedoch explizit aufrufen kann. Wenn dieselbe Funktion mehrmals redefiniert wird, überschreibt jede neue Definition die vorige, kann diese bei Bedarf jedoch aufrufen, so dass eine *lineare Kette* von *Zweigen* entsteht (vgl. Abb. 7). Durch eine *bedingte Redefinition* kann außerdem vereinbart werden, dass ein Zweig nur ausgeführt wird, wenn die mit ihm assoziierte Bedingung erfüllt ist, während der Aufruf andernfalls automatisch zum vorigen Zweig delegiert wird. Somit wird beim Aufruf einer globalen virtuellen Funktion durch einen Klienten immer der letzte Zweig der Kette aufgerufen, der den Aufruf entweder explizit oder implizit weiterreichen kann.

3.2.2 Dynamisches Binden und strukturelle Erweiterungen

Da man in bedingten Redefinitionen beliebige Eigenschaften der Funktionsargumente überprüfen kann, insbesondere ihren dynamischen Typ, lässt sich mit diesem relativ einfachen Mechanismus sowohl einfaches objektorientiertes als auch mehrfaches und prädikatbasiertes *dynamisches Binden*

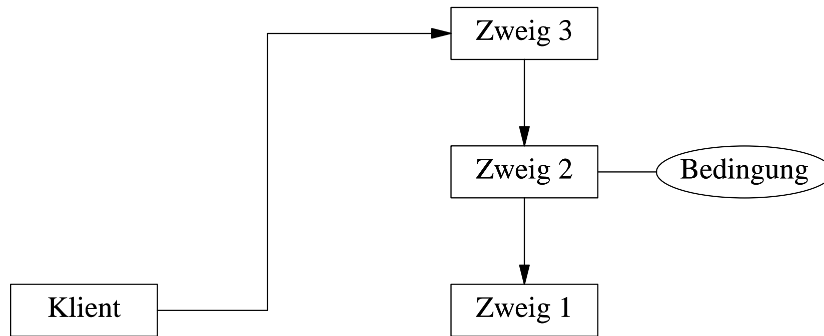

Abbildung 7: Globale virtuelle Funktion mit drei Zweigen

nachbilden. Insbesondere treten keinerlei spezielle Probleme mit *binären Methoden* auf, d. h. mit Operationen, die auf mehreren Objekten desselben Typs operieren.

Um dies exemplarisch zu illustrieren, zeigt Abb. 8 zunächst die Definition eines offenen Typs `Expr` mit zugehörigen Attributen und benutzerdefinierten Konstruktoren zur Repräsentation unterschiedlicher Arten arithmetischer Ausdrücke. Zur Auswertung von Ausdrücken, d. h. zur Berechnung ihres

```
// Offener Typ mit zugehörigen Attributen zur Repräsentation
// unterschiedlicher Arten arithmetischer Ausdrücke.
typename Expr;
Expr -> integer value;  // Wert eines konstanten Ausdrucks.
Expr -> character oper; // Operator eines binären Ausdrucks.
Expr -> Expr left;      // Linker und rechter Teilausdruck
Expr -> Expr right;     // eines binären Ausdrucks.

// Benutzerdefinierte Konstruktoren zur Erzeugung
// konstanter bzw. binärer Ausdrücke.
Expr (integer v) {
    return Expr(@value, v);
}
Expr (Expr l, character o, Expr r) {
    return Expr(@left, l)(@oper, o)(@right, r);
}

// Gloable virtuelle Funktion mit mehreren Zweigen
// zur Auswertung von Ausdrücken.
virtual integer eval (Expr x) if (x@value) {
    return x@value;
}
virtual integer eval (Expr x) if (x@oper == '+') {
    return eval(x@left) + eval(x@right);
}
virtual integer eval (Expr x) if (x@oper == '-') {
    return eval(x@left) - eval(x@right);
}
...... // Analog für '*' und '/'.
```

Abbildung 8: Repräsentation, Erzeugung und Auswertung arithmetischer Ausdrücke

Werts, wird eine globale virtuelle Funktion `eval` definiert, die für jede Ausdrucksart (konstante Ausdrücke, Additionen, Subtraktionen etc.) einen passenden bedingten Zweig definiert. Ein konkreter Funktionsaufruf `eval(x)` gelangt zunächst zum letzten Zweig der Funktion und überprüft dessen assoziierte Bedingung. Wenn sie erfüllt ist, wird sein Rumpf ausgeführt, während der Aufruf andernfalls zum vorigen Zweig delegiert wird, der nach demselben Prinzip verfährt. Auf diese Weise wird zur Laufzeit abhängig von den Attributwerten des konkreten Objekts `x` jeweils der passende Zweig ausgeführt (dynamisches Binden).[2] Sollte das Spektrum der Operatoren binärer Ausdrücke später erweitert werden (z. B. um einen Modulo-Operator `%`), kann jederzeit ein weiterer passender Zweig von `eval` ergänzt werden, ohne den Code der bereits geschriebenen Zweige verändern zu müssen. Ebenso ist es möglich, vollkommen neue Arten von Ausdrücken (wie z. B. unäre Ausdrücke) hinzuzufügen, indem man zusätzliche Attribute für den Typ `Expr` definiert (z. B. `body` zur Speicherung des Rumpfs eines unären Ausdrucks), geeignete Konstruktoren ergänzt und zusätzliche Zweige für `eval` definiert (vgl. Abb. 9). Dies entspricht in etwa dem Hinzufügen einer neuen Unterklasse inklusive Konstruktoren und überschreibenden Methoden zu einer bestehenden objektorientierten Klassenhierarchie.

```
Expr -> Expr body;       // Rumpf eines unären Ausdrucks.

// Konstruktor zur Erzeugung eines unären Ausdrucks.
Expr (character o, Expr b) {
    return Expr(@oper, o)(@body, b);
}

// Zusätzlicher Zweig von eval zur Auswertung unärer Ausdrücke.
virtual integer eval (Expr x) if (x@body) {
    if (x@oper == '+') return eval(x@body);
    if (x@oper == '-') return -eval(x@body);
}
```

Abbildung 9: Modulare strukturelle Erweiterung um unäre Ausdrücke

3.2.3 Operationale Erweiterungen

Da globale virtuelle Funktionen unabhängig voneinander in unterschiedlichen Modulen definiert werden können, ist es problemlos möglich, nachträglich weitere Operationen zu einem bestehenden System hinzuzufügen, ohne den vorhandenen Quellcode ändern oder neu übersetzen zu müssen, d. h. das *Erweiterbarkeitsproblem* existiert nicht.

Zur Illustration zeigt Abb. 10 die Definition einer weiteren globalen virtuellen (Operator-)Funktion `operator==`, die implizit für Vergleiche `x == y` zweier `Expr`-Objekte `x` und `y` aufgerufen wird. Da für die Auswahl des passenden Zweigs die Attributwerte *beider* Parameter relevant sind, werden diese in den assoziierten Bedingungen entsprechend überprüft, d. h. es wird dynamisches Binden anhand mehrerer Objekte (*multiple dispatch*) angewandt.

3.2.4 Verhaltenserweiterungen

Schließlich können Redefinitionen globaler virtueller Funktionen auch dazu verwendet werden, *crosscutting concerns* wie z. B. Synchronisation oder Protokollierung unabhängig von der Kernfunk-

[2] Die Zweige für binäre Ausdrücke gehen davon aus, dass Objekte `x`, die einen Operator `x@oper` besitzen, auch Teilausdrücke `x@left` und `x@right` besitzen. Bei ausschließlicher Verwendung der zuvor definierten benutzerdefinierten Konstruktoren zur Erzeugung von `Expr`-Objekten ist diese Annahme auch zutreffend. Um dies zu erzwingen, d. h. direkte Aufrufe des Attribut-Initialisierungs-Konstruktors in Klientenmodulen zu verbieten, kann der Typ `Expr` *halböffentlich* definiert werden (vgl. Abschnitt 3.3.4).

```
// Globale virtuelle (Operator-)Funktion zum Vergleich von Ausdrücken.
virtual bool operator== (Expr x, Expr y) {
    return false;
}
virtual bool operator== (Expr x, Expr y) if (x@value && y@value) {
    return x@value == y@value;
}
virtual bool operator== (Expr x, Expr y) if (x@oper && y@oper) {
    return x@oper == y@oper && x@left == y@left && x@right == y@right;
}
virtual bool operator== (Expr x, Expr y) if (x@body && y@body) {
    return x@oper == y@oper && x@body == y@body;
}
```

Abbildung 10: Operationale Erweiterung zum Vergleich arithmetischer Ausdrücke

tionalität eines Systems in separaten Modulen zu implementieren, d. h. sie decken das Konzept von *advice* in aspektorientierten Programmiersprachen ohne zusätzliche Sprachmittel mit ab.

Abbildung 11 zeigt als einfaches Beispiel eine Redefinition von `eval`, die zunächst mit Hilfe des Pseudo-Funktionsnamens `virtual` den vorigen Zweig der Funktion aufruft, um den Ausdruck `x` auszuwerten; anschließend wird das Ergebnis `v` des Aufrufs protokolliert und zurückgeliefert. Man beachte, dass die Pseudofunktion `virtual` grundsätzlich ohne explizite Argumente aufgerufen wird, da die aktuellen Argumente des Funktionsaufrufs implizit an den vorigen Zweig weitergereicht werden.

```
// Redefinition von eval zur Protokollierung von Aufrufen.
virtual integer eval (Expr x) {
    // Vorigen Zweig der Funktion aufrufen.
    integer v = virtual();

    // Aufruf protokollieren.
    cout << "eval: " << v << endl;

    // Ergebnis zurückliefern.
    return v;
}
```

Abbildung 11: Verhaltenserweiterung zur Protokollierung der Aufrufe von `eval`

3.2.5 Weiterführende Information

Wenn der erste Zweig einer globalen virtuellen Funktion explizit oder implizit seinen vorigen Zweig aufruft, wird ein automatisch erzeugter *Zweig 0* aufgerufen, dessen Rumpf leer ist und einen Nullwert des entsprechenden Resultattyps zurückliefert [3, 5]. Daher ist ein Funktionsaufruf auch dann wohldefiniert, wenn keine der mit den Zweigen assoziierten Bedingungen erfüllt ist.

Da benutzerdefinierte *Konstruktoren* eines offenen Typs ebenfalls globale Funktionen sind, können auch sie virtuell deklariert und so bei Bedarf nachträglich redefiniert werden [1]. Außerdem ist jedes *Attribut* eines offenen Typs äquivalent zu einem Paar globaler virtueller Funktionen (einer *Lese-* und einer *Schreibfunktion*), die implizit bei jedem lesenden bzw. schreibenden Attributzugriff aufgerufen werden [1]. Durch geeignete Redefinitionen dieser Funktionen lassen sich z. B. Attributzugriffe automatisch protokollieren oder persistent gespeicherte Attributwerte implementieren, indem die Lese-

funktion den aktuellen Wert des Attributs z. B. aus einer Datei liest, während die Schreibfunktion den neuen Wert dorthin schreibt.

Schließlich gibt es *lokale virtuelle Funktionen* [4], d. h. *temporäre* Redefinitionen globaler virtueller Funktionen für die Dauer eines Anweisungsblocks, mit denen sich z. B. *control flow join points* aspektorientierter Sprachen sowie *Ausnahmebehandlung* (sogar mit der Möglichkeit von *resumption*, d. h. Wiederaufsetzen an der Fehlerstelle) nachbilden lassen, wiederum ohne hierfür spezialisierte Sprachmittel (wie z. B. `try`, `catch` und `throw`) verwenden zu müssen.

Veröffentlichung [5], die eine erweiterte Fassung des Konferenzbeitrags [3] darstellt, enthält neben Kapitel 5 des Vorlesungsskripts [11] die aktuellste und umfassendste Darstellung des Konzepts globaler virtueller Funktionen. Anhand von Beispielen wird gezeigt, dass mit diesem relativ einfachen Mechanismus sowohl einfaches, mehrfaches und prädikatbasiertes dynamisches Binden objektorientierter Sprachen als auch *advice* aspektorientierter Sprachen abgedeckt wird. Insbesondere wird eine einfache Lösung des *expression problems* präsentiert. Außerdem werden lokale virtuelle Funktionen als naheliegende Erweiterung des Konzepts vorgestellt, die das Anwendungsspektrum nochmals wesentlich erweitern. Schließlich werden die wesentlichen Ideen zur Implementierung der Konzepte erläutert.

Veröffentlichung [4] konzentriert sich auf lokale virtuelle Funktionen und *nicht-lokale Sprunganweisungen* und stellt typische Anwendungsbeispiele für diese erweiterten Konzepte vor. Insbesondere wird gezeigt, wie damit Ausnahmebehandlung mit der Möglichkeit des Wiederaufsetzens an der Ausnahmestelle realisiert werden kann, ohne hierfür einen speziellen Ausnahmebehandlungsmechanismus (wie z. B. `try`, `catch` und `throw`) verwenden zu müssen.

Veröffentlichung [2] schließlich zeigt durch eine Einbettung globaler virtueller Funktionen in Java, dass das Konzept tatsächlich sprachunabhängig ist und problemlos in unterschiedliche Sprachen integriert werden kann.

Alle genannten Veröffentlichungen enthalten wiederum ausführliche Diskussionen verwandter Arbeiten, insbesondere Multimethoden in Sprachen wie CLOS [26], Dylan [17] und MultiJava [16, 30] sowie *advice* in aspektorientierten Sprachen wie AspectJ [27] und AspectC++ [33].

Der wesentliche Unterschied zwischen globalen virtuellen Funktionen und Multimethoden besteht darin, dass bei letzteren die Definitionsreihenfolge der Methoden irrelevant ist und stattdessen bei jedem Aufruf die *am besten* passende Definition ausgewählt wird. Obwohl dies auf den ersten Blick vorteilhaft erscheint, verursacht dieser Ansatz in der Praxis diverse Probleme, insbesondere wenn statisch sichergestellt werden soll, dass eine über mehrere Module verteilte Menge von Methodendefinitionen sowohl *vollständig* als auch *eindeutig* ist, d. h. dass es zu jedem denkbaren Methodenaufruf *genau eine* am besten passende Definition gibt. Außerdem muss die Palette der Auswahlkriterien zwangsläufig eingeschränkt werden, typischerweise auf dynamische Typtests und eventuell Tests auf individuelle Werte, da die Frage nach der am besten passenden Definition bei uneingeschränkten Auswahlprädikaten nicht entscheidbar ist. Im Gegensatz dazu können die Zweige einer globalen virtuellen Funktion tatsächlich beliebige Auswahlprädikate besitzen, von denen zur Laufzeit jeweils das *erste* passende in umgekehrter Definitionsreihenfolge ausgewählt wird. Durch geeignete Anordnung der Zweige – die sich in vielen Fällen von selbst richtig ergibt, indem man spezifischere Definitionen nach allgemeineren plaziert, ebenso wie man in objektorientierten Sprachen abgeleitete Klassen zwangsläufig nach ihren Basisklassen definiert – kann man so einerseits Multimethoden perfekt nachbilden und hat andererseits noch wesentlich mächtigere Auswahlkriterien zur Verfügung.

Anders als in aspektorientierten Sprachen, wo gewöhnliche Funktionen bzw. Methoden einerseits und *advice* zu ihrer modularen Erweiterung andererseits getrennte Konzepte darstellen, vereinigen globale virtuelle Funktionen diese beiden Aspekte in einem einzigen einfachen Konzept. Außerdem besteht, wie bereits in Abschnitt 2.2 erwähnt, kein Unterschied zwischen ursprünglich vorhandenen und nachträglich hinzugefügten Funktionen.

Kapitel 5 des Vorlesungsskripts [11] bietet ebenfalls eine gut verständliche Einführung in die wesentlichen Konzepte globaler und lokaler virtueller Funktionen.

3.3 Module

3.3.1 Konzept

Das Konzept von Modulen wurde weitgehend von Modula-2 [40] und Oberon [41] übernommen, wo Module primär zur *Gruppierung* logisch zusammengehörender Typen und Funktionen verwendet werden. Durch die Unterscheidung öffentlicher und privater Definitionen ist es darüber hinaus möglich, Datenkapselung im Sinne des *Geheimnisprinzips* [32] zu erreichen. Außerdem stellt jedes Modul einen unabhängigen *Namensbereich* dar, so dass gleichnamige Typen, Attribute oder Funktionen in unterschiedlichen Modulen voneinander unabhängig sind und nicht miteinander in Konflikt stehen. Schließlich können Module separat übersetzt und auf statische Typsicherheit überprüft werden.

3.3.2 Initialisierungs- und Aktivierungsreihenfolge

Um die öffentlichen Definitionen eines Moduls A in einem anderen Modul B benutzen zu können, muss A als *Basismodul* von B angegeben werden. Aus diesen *Abhängigkeitsbeziehungen* zwischen Modulen lässt sich durch topologische Sortierung eine eindeutige *Initialisierungsreihenfolge* für alle Module eines Programms ableiten, die gewährleistet, dass die Basismodule eines Moduls vollständig initialisiert sind, bevor mit der Initialisierung des Moduls selbst begonnen wird. Dadurch ist insbesondere sichergestellt, dass beim Aufruf einer Funktion aus einem Basismodul sämtliche Datenstrukturen initialisiert sind, auf die diese Funktion möglicherweise zugreift.

Darüber hinaus impliziert die genannte Initialisierungsreihenfolge von Modulen eine entsprechende eindeutige *Aktivierungsreihenfolge* für die Zweige globaler virtueller Funktionen, sofern diese über mehrere Module verteilt sind [5, 1]. Insbesondere werden die in einem bestimmten Modul definierten Zweige *nach* den Zweigen aktiviert, die in seinen Basismodulen definiert sind, d. h. sie werden *nach* diesen in die in Abschnitt 3.2.1 erwähnte Kette von Zweigen eingefügt.

Beispielsweise induzieren die folgenden Moduldefinitionen den in Abb. 12 gezeigten Abhängigkeitsgraphen, dessen topologische Sortierung die eindeutige Initialisierungsreihenfolge `expr` →

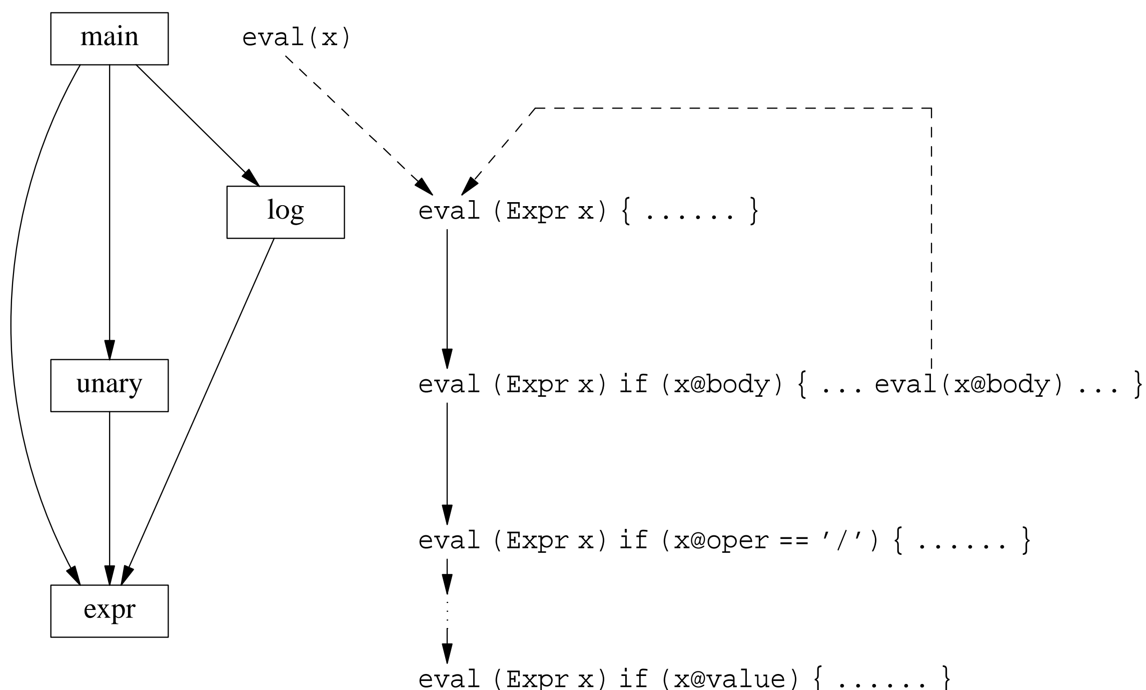

Abbildung 12: Abhängigkeitsgraph von Modulen und Aktivierungsreihenfolge von Funktionszweigen

unary → log → main impliziert (im Zweifelsfall ist die textuelle Reihenfolge, in der die Basismodule eines Moduls angegeben werden, ausschlaggebend):

```
// Basismodul expr enthält die Definitionen aus Abb. 8.
namespace expr { ...... }

// Abhängiges Modul unary enthält die Definitionen aus Abb. 9.
namespace unary : expr { ...... }

// Abhängiges Modul log enthält die Definitionen aus Abb. 11.
namespace log : expr { ...... }

// Hauptmodul main.
namespace main : expr, unary, log { ...... }
```

Daher werden die im Modul expr definierten Zweige der Funktion eval als erstes aktiviert und stehen daher am Anfang der entsprechenden Kette. (Innerhalb eines Moduls ist die textuelle Reihenfolge der Zweige maßgebend.) Als nächstes folgt der in Modul unary definierte Zweig und schließlich der in Modul log definierte (vgl. Abb. 12). Daher gelangt ein Funktionsaufruf eval(x), der beispielsweise im Hauptmodul main ausgeführt wird, zunächst zum Zweig in log. Da dieser keine assoziierte Bedingung besitzt, wird sein Rumpf in jedem Fall ausgeführt. Der darin enthaltene explizite Aufruf des vorigen Zweigs gelangt zunächst zum Zweig in unary. Sofern x ein unärer Ausdruck ist, d. h. x@body nicht null ist, wird dessen Rumpf ebenfalls ausgeführt, während der Aufruf andernfalls an den letzten Zweig in expr delegiert wird usw. Tatsächlich gilt der soeben beschriebene Ablauf für sämtliche Aufrufe von eval im gesamten Programm, insbesondere auch für rekursive Aufrufe der Funktion in einem ihrer Zweige (vgl. Abb. 12).

3.3.3 Dynamisch geladene Module

Da Module separat übersetzt und auf statische Typsicherheit überprüft werden können, kann beim Zusammenlinken von Modulen zu einem Programm kein Fehler mehr auftreten. Daher ist es auch möglich, Module erst zur Laufzeit eines Programms *dynamisch* zu laden (und ggf. wieder zu entladen), wodurch sich zusätzliche interessante Möglichkeiten ergeben. Insbesondere kann ein offener Typ auf diese Weise dynamisch um zusätzliche Attribute erweitert werden, und globale virtuelle Funktionen können zusätzliche Zweige erhalten, die ihr Verhalten erweitern oder modifizieren können [1].

Allerdings sind die in einem dynamisch geladenen Modul definierten Attribute statisch nur in diesem Modul (und, sofern es sich um öffentliche Definitionen handelt, in davon abhängigen Modulen) bekannt und können daher auch nur dort verwendet werden. Insbesondere erfährt der Code des ursprünglichen Programms nicht einmal von ihrer Existenz. Um einem Objekt des entsprechenden offenen Typs Werte für diese Attribute zuweisen zu können, ist es daher notwendig, dass der Code des dynamisch geladenen Moduls eine Referenz auf das Objekt erhält, über die er es verändern kann. Dies geschieht typischerweise dadurch, dass eine solche Referenz als aktueller Parameter an eine Funktion übergeben wird, die im dynamisch geladenen Modul redefiniert wird.

Abbildung 13 illustriert diesen vermeintlich komplizierten Sachverhalt an einem einfachen Beispiel. Das dort gezeigte Modul caching definiert einerseits ein zusätzliches Attribut cache für den im Modul expr definierten Typ Expr und andererseits eine Redefinition der ebenfalls dort definierten globalen virtuellen Funktion eval. Wenn dieses Modul dynamisch von einem Programm geladen wird, erhält der Typ Expr das zusätzliche Attribut und die Funktion eval einen zusätzlichen Zweig, der am Ende ihrer Kette angefügt wird. Allerdings besitzt keines der bis jetzt erzeugten Objekte des Typs Expr einen Wert für das neue Attribut, d. h. entsprechende Zugriffe liefern einen Nullwert. Wenn nun für eines dieser Objekte die Funktion eval(x) aufgerufen wird, gelangt der Aufruf zu dem soeben dynamisch geladenen Zweig im Modul caching, der auf diese Weise die Möglichkeit erhält, den Wert des Attributs für dieses Objekt abzufragen und ggf. zu setzen. Konkret wird dort geprüft, ob

```
// Von expr abhängiges Modul caching.
namespace caching : expr {
    // Verwendung des Namens Expr des Moduls expr.
    using expr::Expr;

    // Zusätzliches Attribut für den Typ Expr.
    Expr -> integer cache;

    // Zusätzlicher Zweig für die Funktion eval.
    virtual integer expr::eval (Expr x) {
        if (!x@cache) x(@cache, virtual());
        return x@cache;
    }
}
```

Abbildung 13: Dynamisch zu ladendes Modul

x@cache bereits einen Wert enthält, und wenn nicht, wird dieser durch einen Aufruf des vorigen Zweigs der Funktion (der den Wert des Ausdrucks x berechnet) ermittelt und gesetzt, so dass beim nächsten Aufruf der Funktion für dasselbe Objekt x dieser gespeicherte Wert geliefert werden kann, ohne den Wert des Ausdrucks erneut berechnen zu müssen.

Obwohl der ursprüngliche Programmcode nichts von der Existenz des Attributs cache weiß, besitzt nun jedes Expr-Objekt, das diesen Prozess einmal durchlaufen hat, einen Wert für dieses Attribut, der immer dann „sichtbar" und verwendbar ist, wenn der Code des Moduls caching über eine Referenz auf dieses Objekt zugreifen kann. Die entsprechenden Attributwerte bleiben sogar erhalten, wenn das Modul caching später wieder entladen wird, was zur Folge hat, dass sein Zweig der Funktion eval wieder deaktiviert, d. h. aus der entsprechenden Kette entfernt wird und Aufrufe der Funktion daher wieder zum ursprünglich letzten Zweig geleitet werden. Allerdings gibt es in diesem Programmzustand nun keinerlei Code mehr, der das Attribut cache statisch kennt und daher darauf zugreifen könnte. Durch ein erneutes Laden des Moduls wird sein Zweig von eval jedoch erneut aktiviert, so dass Aufrufe der Funktion jetzt wieder dorthin gelangen. Beim Zugriff auf x@cache findet der entsprechende Code dann die früher von ihm gesetzten Attributwerte unverändert vor.

Wie dieses Beispiel zeigt, können durch das dynamische Laden und Entladen von Modulen bestimmte „Aspekte" eines Programms (wie z. B. Caching, aber auch Synchronisation, Protokollierung o. ä.) bequem dynamisch ein- und ausgeschaltet werden. Abgesehen von dem einfachen und allgemein nützlichen Mechanismus zum dynamischen Laden und Entladen von Code, werden auch hierfür wieder keinerlei spezialisierte Sprachmittel benötigt. (Und im Gegensatz zu *dynamischer Aspektorientierung*, wo derartige Möglichkeiten als ausgesprochen schwierig zu realisieren eingestuft werden, ist auch die zugrundeliegende Implementierung vergleichsweise einfach.)

3.3.4 Weiterführende Information

Neben öffentlichen und privaten Definitionen können Module auch *halböffentliche* Definitionen enthalten, um beispielsweise Attribute offener Typen für andere Module nur lesbar, aber nicht änderbar zur Verfügung zu stellen oder globale virtuelle Funktionen in anderen Modulen nur redefinieren, aber nicht direkt aufrufen zu können [1].

Zusätzlich zur Initialisierung globaler Variablen und zur Aktivierung der Zweige globaler virtueller Funktionen, können während der Initialisierung eines Moduls beliebige Code-Blöcke (sog. *Initialisierungsblöcke*) ausgeführt werden. Insbesondere übernehmen die Initialisierungsblöcke des Hauptmoduls die Rolle des *Hauptprogramms*, für das in C++ und Java eine spezielle Funktion main benötigt wird.

Komplementär zur Initialisierung von Modulen, werden am Ende einer Programmausführung alle Module *terminiert*, indem für ihre globalen Variablen ggf. *Destruktoren* ausgeführt werden, die Zwei-

ge globaler virtueller Funktion *deaktiviert* werden und eventuell vorhandene *Terminierungsblöcke* ausgeführt werden. All diese Aktionen erfolgen in umgekehrter Reihenfolge der entsprechenden Initialisierungsschritte. Die Ausführung eines Programms besteht somit aus der Initialisierung aller seiner Module (in der in Abschnitt 3.3.2 genannten Reihenfolge), gefolgt von ihrer Terminierung (in der umgekehrten Reihenfolge).

Beim Entwurf von Modulen nach dem *Geheimnisprinzip* ist es ratsam, die öffentliche Schnittstelle eines Moduls möglichst klein zu halten, d. h. möglichst viele Dinge privat zu definieren, damit abhängige Module von Änderungen dieser Dinge nicht betroffen sind [32]. Auf der anderen Seite ist es mit Blick auf das in dieser Arbeit propagierte *Erweiterbarkeitsprinzip* ratsam, möglichst viele Dinge öffentlich (oder zumindest halböffentlich) zu definieren, damit andere Module bei Bedarf Erweiterungen oder Redefinitionen vornehmen können. Somit stehen diese beiden fundamentalen Prinzipien, die beide ihre Berechtigung haben, mehr oder weniger stark in Konflikt zueinander. Als Ausweg aus diesem Dilemma besteht die Möglichkeit, bei der Definition eines abhängigen Moduls nicht nur die öffentliche Schnittstelle eines Basismoduls, sondern auch dessen *private* Definitionen zu verwenden, wenn dies entsprechend gekennzeichnet wird. Allerdings macht sich das abhängige Modul damit auch von allen Interna des Basismoduls abhängig, so dass sein Code bei Änderungen des Basismoduls ggf. mitgeändert werden muss, d. h. logisch ist so ein abhängiges Modul fast ein Teil des entsprechenden Basismoduls. Nichtsdestotrotz kann es separat geschrieben und übersetzt werden, was insbesondere dann vorteilhaft ist, wenn der Quellcode des Basismoduls nicht vorliegt oder aus anderen Gründen nicht verändert werden soll.

Mit dieser Möglichkeit als „Hintertür“ kann man beim Entwurf eines Moduls im Zweifelsfall immer das Geheimnisprinzip höher achten als das Erweiterbarkeitsprinzip und bei späteren modularen Erweiterungen notfalls auf den privaten Teil des Moduls zugreifen, sofern man es nicht direkt verändern will oder kann. Natürlich sollte diese Möglichkeit nicht als pauschaler „Freibrief“ betrachtet werden, das Geheimnisprinzip nach Belieben zu umgehen, sondern vielmehr als „Legitimation“ für Ausnahmefälle, in denen man keine andere Wahl hat (so wie ein Arzt in Notfällen bestimmte Maßnahmen ergreifen darf, für die er normalerweise eine schriftliche Einwilligung des Patienten braucht). Auch sollte man die Möglichkeit, auf private Dinge eines Moduls zuzugreifen, nicht primär als „Recht“ betrachten, das man sich selbst willkürlich verschafft, sondern vielmehr als „Bindung“ an diese Dinge, der man sich freiwillig unterwirft (d. h. man macht sich eher zum „Sklaven“ als zum „Herrn“ eines solchen Moduls).

Die wesentlichen Konzepte von Modulen werden in den Veröffentlichungen [5] und [1] beschrieben, jeweils im Zusammenhang mit globalen virtuellen Funktionen und (in [1]) offenen Typen. Kapitel 7 des Vorlesungsskripts [11] bietet ebenfalls eine gut verständliche Einführung in die wesentlichen Konzepte von Modulen.

4 Praktische Umsetzung der Konzepte

4.1 Die Programmiersprache C+++

Wie bereits erwähnt, wurden alle im Rahmen dieser Arbeit entwickelten Konzepte exemplarisch als Spracherweiterungen für C++ implementiert [1, 5]. Die resultierende Programmiersprache C+++ wurde und wird bereits in vorlesungsbegleitenden Übungen, Praktika und Diplomarbeiten praktisch eingesetzt. Außerdem ist der Präcompiler, der zur Transformation von C+++ nach C++ verwendet wird, selbst in C+++ geschrieben. Er basiert auf einem kommerziellen C++ Front End der Firma Edison Design Group (`www.edg.com`), das für Forschungszwecke freundlicherweise kostenlos zur Verfügung gestellt wurde. Durch einen kleinen technischen Trick war es möglich, sämtliche Funktionen dieses ca. 400.000 Zeilen C-Code umfassenden Programms in globale virtuelle Funktionen umzuwandeln, um sie anschließend nach Belieben redefinieren zu können. Und tatsächlich sind alle notwendigen Erweiterungen und Modifikationen dieses Programms streng modular und *nicht-invasiv* aus-

schließlich durch Redefinitionen der vorhandenen Funktionen implementiert worden, ohne eine einzige Zeile des ursprünglichen Codes verändern oder neu übersetzen zu müssen. Damit konnte anhand eines realen Projekts demonstriert werden, dass die hier vorgeschlagenen Konzepte zur modularen Erweiterbarkeit bestehender Software-Systeme tatsächlich praktikabel sind, selbst wenn sie bei der Entwicklung des ursprünglichen Systems überhaupt nicht berücksichtigt wurden.

Neben diesem Präcompiler gibt es ein Laufzeitsystem, das sich um die effiziente Speicherverwaltung und -bereinigung für Objekte offener Typen kümmert. Hierbei wurde insbesondere darauf geachtet, dass für die zusätzliche Flexibilität offener Typen kein allzu hoher Laufzeitpreis zu bezahlen ist. Erste Performance-Messungen haben ergeben, dass C+++-Programme etwa 1,9- bis 2,5-mal langsamer als funktional äquivalente C++-Programme sind, was nicht überrascht, wenn man bedenkt, dass C bzw. C++ häufig als Performance-Messlatte angesehen wird. Andererseits sind C+++-Programme bereits jetzt bis zu 2,5-mal schneller als funktional äquivalente Java-Programme auf einer virtuellen Maschine mit Just-in-time-Compiler [1].

4.2 Speicherverwaltung für Attribute offener Typen

Abbildung 14 illustriert die wesentlichen Ideen zur Implementierung von Objekten offener Typen, die Werte für unterschiedliche Teilmengen von Attributen besitzen [1]. Der *Wert eines offenen Typs* ist hierbei ein Zeiger auf eine *Hilfszelle*, die ihrerseits auf einen zugehörigen *Datenbereich* verweist, der die Attributwerte des Objekts enthält (absteigend sortiert nach ihren Ausrichtungsanforderungen, um Lücken zwischen den Werten zu vermeiden). Außerdem verweist die Hilfszelle auf einen *Objektdeskriptor*, der den Inhalt des Datenbereichs beschreibt und der von allen Objekten gemeinsam verwendet wird, deren zugehörige Datenbereiche gleich strukturiert sind. Für jedes Attribut des offenen Typs (im Beispiel `Expr`) enthält ein Objektdeskriptor die Information, ob und ggf. an welcher Position der entsprechende Attributwert im Datenbereich eines Objekts gespeichert ist: Wenn das zugehörige Kästchen eine Zahl enthält, gibt sie die relative Adresse des Attributwerts im Datenbereich an, während ein leeres graues Kästchen das Fehlen des Attributs anzeigt.

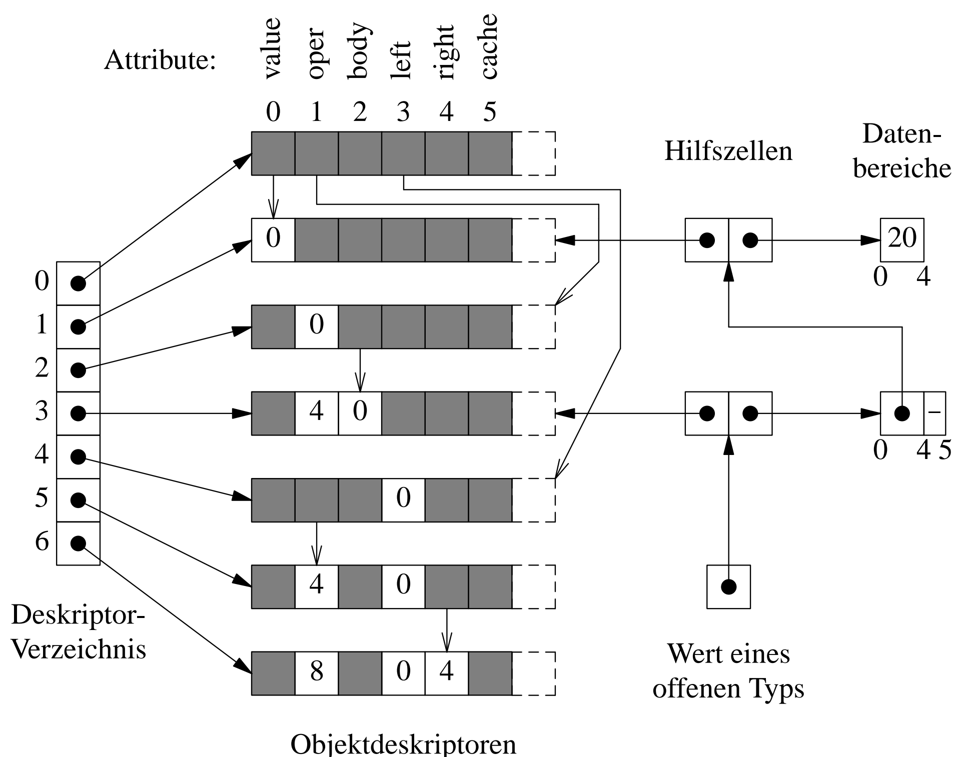

Abbildung 14: Speicherverwaltung für Attribute

Basierend auf dieser Information, läuft ein Attributzugriff nun wie folgt ab: Mit Hilfe einer für jedes Attribut eindeutigen Nummer (z. B. `1` für `oper`) wird auf das entsprechende Kästchen des Objektdeskriptors zugegriffen und der dort stehende Wert als relative Adresse im Datenbereich des Objekts interpretiert, um auf den dort befindlichen Attributwert zuzugreifen. Falls das Kästchen jedoch leer ist, liefert ein lesender Zugriff auf das Attribut einfach den Nullwert seines Zieltyps, während ein schreibender Zugriff den Datenbereich des Objekts vergrößern muss, um Platz für den neuen Wert zu schaffen. (Dabei kann sich die Adresse des Datenbereichs ändern, aber die Adresse der Hilfszelle, die in den Werten des offenen Typs gespeichert wird, bleibt unverändert.) Außerdem muss der Deskriptorzeiger des Objekts auf den „Nachfolger-Deskriptor" umgeleitet werden, der exakt dieselben Attribute wie der aktuelle Deskriptor plus das neue Attribut enthält.

Da Objektdeskriptoren dynamisch bei Bedarf erzeugt werden, kann es sein, dass dieser erweiterte Deskriptor noch nicht existiert. In diesem Fall wird er neu erzeugt und in das *Deskriptorverzeichnis* des Typs eingetragen. Außerdem wird er mit allen bereits vorhandenen Deskriptoren verglichen, um sowohl seine „Nachfolger" als auch seine „Vorgänger" zu finden, d. h. diejenigen Deskriptoren, die genau ein Attribut mehr bzw. weniger besitzen. Diese Vorgänger/Nachfolger-Beziehungen werden gespeichert, indem in das Kästchen eines Vorgängers, das dem zusätzlichen Attribut eines Nachfolgers entspricht, ein Verweis auf diesen Nachfolger eingetragen wird (vgl. die Pfeile zwischen den Deskriptoren in Abb. 14). Mit Hilfe dieser Information kann ein bereits vorhandener Nachfolger-Deskriptor später einfach und effizient gefunden werden.

Typischerweise muss ein Objekt während seiner Initialisierung mit Werten einige Male reorganisiert werden, bis die von ihm benötigte Teilmenge von Attributen vorhanden ist; anschließend bleibt diese Menge (aber natürlich nicht unbedingt die zugehörigen Attributwerte) in der Regel konstant. Ebenso muss das „Netzwerk" von Deskriptoren eines bestimmten Typs während der Initialisierungsphase eines Programms sukzessive aufgebaut werden, bis alle benötigten Deskriptoren inklusive ihrer Vorgänger/Nachfolger-Beziehungen erzeugt sind, über die dann die meisten für Objektreorganisationen benötigten Deskriptoren sofort gefunden werden.

Wenn ein Attribut eines offenen Typs dynamisch geladen wird (beispielsweise `cache`), müssen alle Deskriptoren des Typs ein zusätzliches Kästchen für dieses Attribut erhalten. Hierfür muss das Deskriptorverzeichnis durchlaufen und jeder Deskriptor entsprechend vergrößert (d. h. ggf. neu beschafft und kopiert) werden. Um auszuschließen, dass hierbei die in den Hilfszellen gespeicherten Deskriptorzeiger verändert werden müssen, wird eine zusätzliche Indirektion benötigt, die in Abb. 14 der Einfachheit halber fehlt. Da das dynamische Laden von Attributen normalerweise nicht sehr häufig vorkommt, ist der Aufwand für diese Vergrößerungen akzeptabel. Um ihn noch weiter zu reduzieren, besitzen Objektdeskriptoren von Anfang an einige zusätzliche Kästchen (in Abb. 14 gestrichelt gezeichnet), die bei Bedarf für dynamisch geladene Attribute verwendet werden können.

4.3 Implementierung globaler virtueller Funktionen

Abbildung 15 illustriert die wesentlichen Ideen zur Implementierung globaler virtueller Funktionen (GVF) [5]. Jeder Zweig einer solchen Funktion wird auf eine gewöhnliche globale Funktion mit einem eindeutigen internen Namen (wie z. B. `f__1`) abgebildet, die ansonsten dieselbe Signatur wie die GVF besitzt. Die Verkettung der Zweige wird über globale Funktionszeiger-Variablen realisiert (kleine Quadrate in der Abbildung): zu jedem Zweig gibt es eine Variable, die die Adresse des vorigen Zweigs enthält. Somit können (explizite und implizite) Aufrufe der Pseudofunktion `virtual` auf entsprechende indirekte Funktionsaufrufe über diese Variable abgebildet werden. In einer weiteren Funktionszeiger-Variablen wird jeweils die Adresse des letzten Zweigs der Kette gespeichert. Schließlich gibt es eine *Verteilerfunktion*, deren Signatur exakt der Signatur der GVF entspricht – d. h. die bei jedem Aufruf der GVF ausgeführt wird –, die normalerweise den letzten Zweig der Kette über diese Variable aufruft. Falls die Kette jedoch nur einen einzigen Zweig enthält, wird dieser von der Verteilerfunktion direkt aufgerufen (gestrichelter Pfeil in der Abbildung), um die Indirektion über die Funktionszeiger-Variable zu vermeiden. Außerdem kann ein optimierender Compiler diesen direkten Aufruf ggf. inline expandieren, um das Laufzeitverhalten zu verbessern. Wenn dann auch noch der Aufruf der Verteilerfunktion (deren Code extrem kurz ist) in einer Klientenfunktion inline expandiert wird, ist

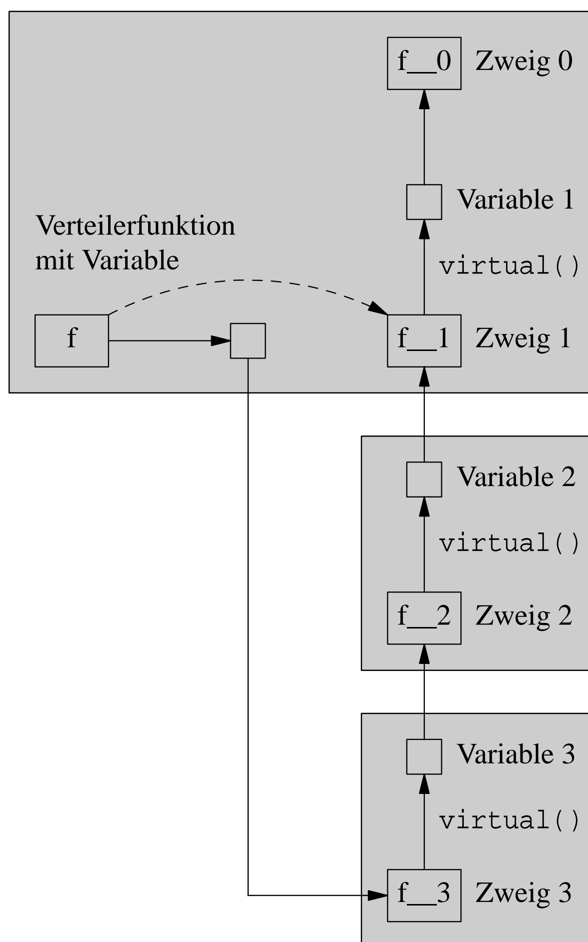

Abbildung 15: Implementierung globaler virtueller Funktionen

für den Aufruf einer GVF, die nur einen Zweig besitzt, kein höherer Laufzeitpreis zu bezahlen als für einen gewöhnlichen Funktionsaufruf. Dies ist insbesondere für die Lese- und Schreibfunktionen von Attributen relevant (vgl. Abschnitt 3.2.5), deren Code relativ kurz ist und die nur selten redefiniert werden.

Bei der Transformation des ersten Zweigs einer GVF wird neben der eigentlichen Zweigfunktion und der zugehörigen Funktionszeiger-Variable auch die Verteilerfunktion mit der zugehörigen Variable sowie der leere Zweig 0 generiert (großer grauer Kasten in Abb. 15), während für die nachfolgenden Zweige jeweils nur die Zweigfunktion und die zugehörige Variable erzeugt werden (kleine graue Kästen).

4.4 Weiterführende Information

Um beim Zugriff auf ein Attribut (vgl. Abb. 14) die Fallunterscheidung zwischen weißen und grauen Kästchen – die zwangsläufig zu einem nicht vorhersagbaren und damit potentiell „teuren" bedingten Sprung im Assembler-Code führt – zu vermeiden, enthält auch ein graues Kästchen in Wirklichkeit eine relative Adresse, über die man – ausgehend von der Adresse des umgebenden Deskriptors – direkt zu einer globalen Variablen gelangt, die den Nullwert des zugehörigen Attributs enthält. Außerdem werden die Farben grau und weiß als Zahlen 0 und 1 kodiert, die als Index in die Hilfszelle eines Objekts verwendet werden können, wenn man diese als zweielementiges Array von Adressen interpretiert. Somit kann man beim Zugriff auf einen Attributwert eines Objekts ohne Fallunterscheidung entweder dem Zeiger auf seinen Deskriptor oder dem Zeiger auf seinen Datenbereich folgen und zu

diesem den Inhalt des jeweiligen Deskriptor-Kästchens addieren, um in beiden Fällen die Adresse des richtigen Attributwerts zu erhalten: Im Fall eines grauen Kästchens gelangt man zu der globalen Variablen, die den Nullwert des Attributs enthält, während man bei einem weißen Kästchen wie zuvor zu der Stelle im Datenbereich des Objekts gelangt, an der sich der individuelle Attributwert befindet. Der hieraus resultierende Assembler-Code besteht nur noch aus 5 bis 7 move-Befehlen.

Für die automatische Speicherbereinigung wurde ein inkrementeller Mark&Sweep-Garbage-Collector basierend auf [18] implementiert, der prinzipiell echtzeitfähig ist, d. h. keinerlei unkontrollierte Unterbrechungen des Anwendungsprogramms verursacht. Alternativ kann der konservative Kollektor von Boehm et al. [12] verwendet werden, der zwar nur eingeschränkt inkrementell funktioniert, aber insgesamt wesentlich effizienter arbeitet, weil er keinerlei Interaktion mit dem Anwendungsprogramm erfordert.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurden *verbesserte prozedurale Programmiersprachen* als eine neue Kategorie imperativer Programmiersprachen entwickelt, die eine höhere Ausdrucksmächtigkeit und Flexibilität als objektorientierte Sprachen aufweisen und dennoch mit wesentlich weniger Sprachmitteln und Konzepten auskommen. Darüber hinaus decken sie einige Vorteile aspektorientierter Sprachen mit ab, ohne hierfür zusätzliche spezialisierte Sprachmittel einführen zu müssen.

Insbesondere ist es möglich,

- mehrfache und wiederholte Vererbung in beliebiger Kombination zu verwenden,
- dynamische Objektevolution mit statischer Typsicherheit zu kombinieren,
- einfaches, mehrfaches und prädikatbasiertes dynamisches Binden nachzubilden,
- bestehende Systeme modular um neue Datenstrukturen und Operationen zu erweitern
- sowie das Verhalten eines Systems nachträglich nicht-invasiv zu erweitern und zu modifizieren.

Der Hauptgrund für die wesentlich höhere Flexibilität liegt darin, dass die beiden Kernkonzepte klassischer prozeduraler Sprachen, Datenstrukturen und Prozeduren, *unabhängig voneinander* zu offenen Typen und globalen virtuellen Funktionen verallgemeinert wurden, anstatt sie wie in objektorientierten Sprachen zu Klassen zusammenzuschnüren. Insbesondere bieten globale virtuelle Funktionen dynamisches Binden *unabhängig von Typhierarchien* als eigenständiges Konzept an. Durch die Hinzunahme eines einfachen Modulkonzepts lässt sich auch das Geheimnisprinzip problemlos umsetzen, so dass verbesserte prozedurale Sprachen tatsächlich keinerlei Nachteile gegenüber objektorientierten Sprachen aufweisen.

Ein herausragendes Merkmal des hier vorgestellten Ansatzes ist, dass er ein sehr hohes Maß an *Dynamik* (z. B. dynamisch geladene Attribute offener Typen, dynamisch geladene Zweige globaler virtueller Funktionen und dynamische Objektevolution), das ansonsten bestenfalls von dynamisch typisierten Sprachen und Systemen angeboten wird, mit einem sehr hohen Grad an *statischer Überprüfbarkeit* kombiniert, wie man sie von statisch typisierten Sprachen gewohnt ist. Eine weitere Besonderheit besteht darin, dass viele Konzepte, für die in gängigen Programmiersprachen spezialisierte Sprachmittel nötig sind (z. B. Vererbung, Subtyp-Polymorphie oder nachträgliche Verhaltenserweiterungen), ebenso einfach mit bereits vorhandenen Mitteln ausgedrückt werden können. Somit besteht ein wesentlicher Beitrag dieser Arbeit nicht nur darin, die Flexibilität und Ausdrucksmächtigkeit imperativer Programmiersprachen entscheidend zu verbessern, sondern auch ihre Komplexität erheblich zu reduzieren.

5.2 Ausblick

Die vorgestellten Konzepte verbesserter prozeduraler Programmiersprachen wurden über mehrere Jahre hinweg entwickelt und in zahlreichen Iterationen sukzessive verbessert und verfeinert, so dass sie mittlerweile ein hohes Maß an Reife erlangt haben. Daher werden sich zukünftige Forschungsarbeiten vor allem darauf konzentrieren, den praktischen Einsatz der Konzepte weiter zu verbessern, zum Beispiel durch:

- Performance-Tuning der C+++-Implementierung, insbesondere für globale virtuelle Funktionen mit vielen Zweigen:
  Da mit globalen virtuellen Funktionen häufig das gewohnte objektorientierte dynamische Binden nachgebildet wird, kann ein optimierender Compiler ähnliche Implementierungstechniken (wie z. B. *virtual function tables*) einsetzen, damit zur Laufzeit effizient der jeweils passende Zweig ausgewählt und aufgerufen werden kann. Eine besondere Herausforderung besteht jedoch darin, dass globale virtuelle Funktionen auch wesentlich flexibleres dynamisches Binden unterstützen und die genannten Techniken daher nur unter bestimmten Bedingungen anwendbar sind.
- Entwicklung von C+++-Bibliotheken, beispielsweise für Ein- und Ausgabe, mathematische Applikationen, Graphikprogrammierung, Datenbankzugriff, parallele und verteilte Programmierung etc.
- Einsatz der Sprache in größeren praktischen Projekten, idealerweise in Zusammenarbeit mit industriellen Partnern, um umfangreichere und umfassendere Erfahrungen bei der Anwendung der Konzepte zu sammeln.
  Konkret wäre es reizvoll, einen vollständigen (C+++-)Compiler in C+++ zu schreiben, bei dem beispielsweise Fehlerbehandlung und Optimierung streng modular vom übrigen Code getrennt sind.

Um die prinzipiell einleuchtende Behauptung, dass die Konzepte verbesserter prozeduraler Programmiersprachen sprachunabhängig sind, auch praktisch zu untermauern, wäre es außerdem lohnenswert, sie konkret in andere Sprachen einzubetten. Interessant wäre für ein solches Vorhaben insbesondere für die Sprache Oberon [41], da sie ebenso wie offene Typen und globale virtuelle Funktionen nach dem Einsteinschen Prinzip „Make it as simple as possible, but not simpler“ entwickelt wurde. Im Gegensatz zu C++, wo durch die Einführung dieser Konzepte ein Großteil der Basissprache überflüssig wird, müsste man in Oberon lediglich Records und Arrays durch offene Typen ersetzen sowie gewöhnliche Prozeduren zu redefinierbaren Prozeduren verallgemeinern (letzteres wurde bereits im Rahmen einer prototypischen Implementierung erledigt); das Modulkonzept könnte unverändert übernommen werden.

Schließlich wäre es natürlich reizvoll, eine vollkommen neue Sprache als reine „Advanced Procedural Programming Language“ zu entwerfen, die keinerlei unnötigen „Ballast“ einer existierenden Sprache mehr besitzt.

6 Veröffentlichungen

6.1 Offene Typen

[1] C. Heinlein: "Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance." *Journal of Object Technology* 6, March/April 2007, 50 pages, to appear.
Umfassende Darstellung des Konzepts offener Typen mit zahlreichen Anwendungsbeispielen, u. a. zu *diamond* und *repeated inheritance* und zu *dynamic object evolution*. Auch globale virtuelle Funktionen und Module werden kurz beschrieben, um das Gesamtbild abzurunden und eine Lösung des *expression problems* vorzustellen. Die wesentlichen Ideen zur Implementierung offener Typen werden ebenfalls vorgestellt.

6.2 Globale und lokale virtuelle Funktionen

[2] C. Heinlein: “Dynamic Class Methods in Java.” In: *Net.ObjectDays 2003. Tagungsband* (Erfurt, Germany, September 2003). tranSIT GmbH, Ilmenau, 2003, ISBN 3-9808628-2-8, 215–229.
Einbettung globaler virtueller Funktionen als *dynamic class methods* in Java. Kritische Diskussion des *visitor patterns* zur Lösung des *expression problems*.
Eine Langfassung ist als Ulmer Informatik-Bericht Nr. 2003-05 verfügbar.

[3] C. Heinlein: “Virtual Namespace Functions: An Alternative to Virtual Member Functions in C++ and Advice in AspectC++.” In: *Proc. 2005 ACM Symposium on Applied Computing (SAC)* (Santa Fe, New Mexico, March 2005), 1274–1281.
Einbettung globaler virtueller Funktionen in C++, die damals noch als *virtual namespace functions* bezeichnet wurden.

[4] C. Heinlein: “Local Virtual Functions.” In: R. Hirschfeld, R. Kowalczyk, A. Polze, M. Weske (eds.): *NODe 2005, GSEM 2005* (Erfurt, Germany, September 2005). Lecture Notes in Informatics P-69, Gesellschaft für Informatik e. V., Bonn, 2005, 129–144.
Erweiterung globaler virtueller Funktionen um lokale virtuelle Funktionen und nicht-lokale Sprunganweisungen, mit denen u. a. Ausnahmebehandlung mit Wiederaufsetzen realisiert werden kann.

[5] C. Heinlein: “Global and Local Virtual Functions in C++.” *Journal of Object Technology* 4 (10) December 2005, 71–93, http://www.jot.fm/issues/issue_2005_12/article4.
Langfassung von [3], das auch Teile von [4] enthält und damit den besten Gesamtüberblick zum Thema globale und lokale virtuelle Funktionen bietet. Auch Implementierungsaspekte werden beleuchtet.

6.3 Sonstiges

[6] C. Heinlein: “Null Values in Programming Languages.” In: H. R. Arabnia (ed.): *Proc. Int. Conf. on Programming Languages and Compilers (PLC’05)* (Las Vegas, NV, June 2005), 123–129.
Erläutert die Nützlichkeit von Nullwerten für sämtliche Typen einer Programmiersprache und skizziert mögliche Implementierungen.

[7] A. Gruler, C. Heinlein: “Exception Handling with Resumption: Design and Implementation in Java.” In: H. R. Arabnia (ed.): *Proc. Int. Conf. on Programming Languages and Compilers (PLC’05)* (Las Vegas, NV, June 2005), 165–171.
Erläutert die Nützlichkeit von Ausnahmebehandlung mit der Möglichkeit des Wiederaufsetzens und schlägt hierfür geeignete Spracherweiterungen für Java vor. Durch die Einführung lokaler virtueller Funktionen und nicht-lokaler Sprunganweisungen (vgl. [4]) wurden diese Spracherweiterungen jedoch überflüssig und daher nicht in C+++ übernommen.

[8] C. Heinlein: “C+++: User-Defined Operator Symbols in C++.” In: P. Dadam, M. Reichert (eds.): *INFORMATIK 2004 – Informatik verbindet. Band 2* (Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e. V.; September 2004; Ulm). Lecture Notes in Informatics P-51, Gesellschaft für Informatik e. V., Bonn, 2004, 459–468.
Erweiterung von C++ um die Möglichkeit, benutzerdefinierte Operatorsymbole einzuführen, die der Sprache C+++ ursprünglich ihren Namen gab.
Eine Langfassung ist als Ulmer Informatik-Bericht Nr. 2004-02 verfügbar.

[9] C. Heinlein: “Safely Extending Procedure Types to Allow Nested Procedures as Values.” In: L. Böszörményi, P. Schojer (eds.): *Modular Programming Languages* (Joint Modular Languages Conference, JMLC 2003; Klagenfurt, Austria, August 2003; Proceedings). Lecture Notes in Computer Science 2789, Springer-Verlag, Berlin, 2003, 144–149.
Vorschlag einer Spracherweiterung für Oberon, die es erlaubt, nicht nur globale, sondern auch lokale

Prozeduren gefahrlos als Parameter an andere Prozeduren zu übergeben. Durch die Einführung lokaler virtueller Funktionen als temporäre Redefinitionen globaler Funktionen (vgl. [4]) wurde die Idee jedoch überflüssig.
Eine Langfassung ist als Ulmer Informatik-Bericht Nr. 2003-07 verfügbar.

[10] C. Heinlein: "Implicit and Dynamic Parameters in C++." In: D. Lightfoot, C. Szyperski (eds.): *Modular Programming Languages* (Joint Modular Languages Conference, JMLC 2006; Oxford, England, September 2006; Proceedings). Lecture Notes in Computer Science 4228, Springer-Verlag, Berlin, 2006, 37–56.
Vorschlag einer Spracherweiterung für C++, die es erlaubt, Parameterwerte von Funktionsaufrufen automatisch aus dem statischen oder dynamischen Aufrufkontext zu ermitteln, um so die Länge aktueller Parameterlisten zu reduzieren. Darüber hinaus stellt der Ansatz eine flexiblere und einfachere Alternative zu „dependent names" in C++-Templates dar.
Es ist geplant, das Konzept in die nächste Version von C+++ zu integrieren.

[11] C. Heinlein: *Programmieren in C++ und C+++*. Vorlesungsskript, Universität Ulm, Sommersemester 2006, http://www.informatik.uni-ulm.de/rs/lehre/ProgCppp2006/skript.pdf.
Enthält in den Kapiteln 3, 5 und 7 eine umfassende Einführung in die Programmiersprache C+++, gegliedert anhand der drei Kernkonzepte Datenstrukturen (offene Typen), Operationen (globale virtuelle Funktionen) und Module. Durch direkte Vergleiche mit den jeweils korrespondierenden C++-Konzepten, werden einerseits die Schwachstellen objektorientierter Sprachen aufgezeigt und andererseits die Vorzüge verbesserter prozeduraler Sprachen anschaulich illustriert.

7 Verwendete Literatur

[12] H. Boehm, M. Weiser: "Garbage Collection in an Uncooperative Environment." *Software—Practice and Experience* 18 (9) September 1988, 807–820.

[13] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick: "Living with CLASSIC: When and How to Use a KL-ONE-Like Language." In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 401–456.

[14] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. T. Leavens, B. Pierce: "On Binary Methods." *Theory and Practice of Object Systems* 1 (3) 1995, 221–242.

[15] P. P. Chen: "The Entity-Relationship Model – Toward a Unified View of Data." *ACM Transactions on Database Systems* 1 (1) March 1976, 9–36.

[16] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein: "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java." In: *Proc. 2000 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '00)* (Minneapolis, MN, October 2000). *ACM SIGPLAN Notices* 35 (10) October 2000, 130–145.

[17] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[18] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens: "On-the-Fly Garbage Collection: An Exercise in Cooperation." *Communications of the ACM* 21 (11) November 1978, 966–975.

[19] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[20] A. Goldberg, D. Robson: *Smalltalk-80. The Language*. Addison-Wesley, Reading, MA, 1989.

[21] J. Gosling, B. Joy, G. Steele, G. Bracha: *The Java Language Specification* (Third Edition). Addison-Wesley, Reading, MA, 2005.

[22] W. Harrison, H. Ossher: "Subject-Oriented Programming (A Critique of Pure Objects)." In: *1993 Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (September 1993), 411–428.

[23] J. L. Keedy, G. Menger, C. Heinlein: "Inheriting from a Common Abstract Ancestor in Timor." *Journal of Object Technology* 1 (1) May/June 2002, 81–106, http://www.jot.fm/issues/issue_2002_05/article2.

[24] J. L. Keedy, C. Heinlein, G. Menger: "Inheriting Multiple and Repeated Parts in Timor." *Journal of Object Technology* 3 (10) November/December 2004, 99–120, http://www.jot.fm/issues/issue_2004_11/article1.

[25] J. L. Keedy, C. Heinlein, G. Menger, M. Evered: "Diamond Inheritance and Attribute Types in Timor." *Journal of Object Technology* 3 (10) November/December 2004, 121–142, http://www.jot.fm/issues/issue_2004_11/article2.

[26] S. E. Keene: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA, 1989.

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: "An Overview of AspectJ." In: J. Lindskov Knudsen (ed.): *ECOOP 2001 – Object-Oriented Programming* (15th European Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001, 327–353.

[28] B. Meyer: *Objektorientierte Softwareentwicklung*. Carl Hanser Verlag, München, 1990.

[29] R. MacGregor: "The Evolving Technology of Classification-Based Knowledge Representation Systems." In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 385–400.

[30] T. Millstein, M. Reay, C. Chambers: "Relaxed MultiJava: Balancing Extensibility and Modular Typechecking." In: *Proc. 2003 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '03)* (Anaheim, CA, October 2003). *ACM SIGPLAN Notices* 38 (11) November 2003.

[31] OMG: *OMG Unified Modeling Language Specification* (Version 1.5). Object Management Group, March 2003.

[32] D. L. Parnas: "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 15 (12) December 1972, 1053–1058.

[33] O. Spinczyk, A. Gal, W. Schröder-Preikschat: "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language." In: J. Noble, J. Potter (eds.): *Proc. 40th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)* (Sydney, Australia, February 2002), 53–60.

[34] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

[35] P. Tarr, H. Ossher, W. Harrison, S. M. Sutton Jr.: "N Degrees of Separation: Multi-Dimensional Separation of Concerns." In: *Proc. 21st Int. Conf. on Software Engineering* (May 1999), 107−119.

[36] M. Torgersen: "The Expression Problem Revisited. Four New Solutions Using Generics." In: M. Odersky (ed.): *ECOOP 2004 – Object-Oriented Programming* (18th European Conference; Oslo, Norway, June 2004; Proceedings). Lecture Notes in Computer Science 3086, Springer-Verlag, Berlin, 2004, 123−143.

[37] D. Ungar, R. B. Smith: "Self: The Power of Simplicity." In: *2nd Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Orlando, FL, October 1987). *ACM SIGPLAN Notices* 22 (12) December 1987, 227−241.

[38] N. Wirth: "The Programming Language PASCAL." *Acta Informatica* 1, 1971, 35−63.

[39] N. Wirth: "Modula: A Language for Modular Multiprogramming." *Software—Practice and Experience* 7, 1977, 3−35.

[40] N. Wirth: *Programming in Modula-2*. Springer-Verlag, 1982.

[41] N. Wirth: "The Programming Language Oberon." *Software—Practice and Experience* 18 (7) July 1988, 671−690.