GOS Gofer Objekt-System

Imperativ Objektorientierte

und Funktionale Programmierung

in einer Sprache vereint*

Bernhard Rumpe
Institut für Informatik
Technische Universität München
http: http://www4.informatik.tu-muenchen.de/~rumpe/
email: rumpe@informatik.tu-muenchen.de

15. Dezember 1995

Zusammenfassung

Dieser Artikel gibt eine informelle Übersicht über die Sprache `GOS`, eine imperative, objektorientierte Programmiersprache in der Tradition von `Eiffel` und `C++`, deren Grunddatentypen in der funktionalen Teilsprache `Gofer` definiert werden können.

`GOS` vereint damit das objektorientierte und das funktionale Programmierparadigma auf eine neue Art und Weise.

1 Motivation

Sowohl objektorientierte als auch funktionale Programmierung erfreuen sich immer größerer Beliebtheit, weil sie gegenüber dem klassischen prozeduralen Programmierstil einige Vorteile bieten.

[0]Dieser Artikel ist erschienen in: Kolloquium Programmiersprachen und Grundlagen der Programmierung, Adalbert Stifter Haus, Alt Reichenau, 11.10.1995, Tiziana Margaria (Hrsg.), MIP Bericht N. 9519, Dez. 1995, Universität Passau, Fakultät für Mathematik und Informatik.

Eine funktionale Sprache wie `Gofer` [JON93] bietet ein ausgefeiltes Typsystem mit der Möglichkeit effektiv Datentypen (mit dem `data`-Konstrukt) und darauf Funktionen (mit Patternmatching) zu definieren. Das funktionale Paradigma bietet aufgrund der sehr abstrakten und damit kompakten Definitionsweise die effektivste Art des Programmierens, wenn damit auch gewisse Nachteile bei der Laufzeit-Effizienz hingenommen werden müssen.

Funktionale Programmierung hat aber massive Schwächen bei der Programmierung von stark interaktiven Programmen. Ein Programm, das Eingaben und Ausgaben vornimmt, arbeitet mit Seiteneffekten. Die funktionale Sprache `ML` [PAU91] durchbricht daher das funktionale Paradigma und erlaubt seiteneffektbehaftete Operationen mittels ihres Referenzkonzepts und verliert damit viele Vorteile funktionaler Programmierung, insbesondere der referentiellen Integrität und der damit verbundenen leichten Verständlichkeit des Programmtexts. Andere Ansätze kodieren E/A-Zustände in Monaden, arbeiten mit Continuations oder modellieren interaktive Programme als Abbildungen von Eingabe- und Ausgabesequenzen. Alle diese Varianten besitzen den Nachteil, daß die Auswertungsstrategie des Interpreters eine wesentliche Rolle für das Programmverhalten spielt. So können funktional äquivalente Programme ein völlig unterschiedliches E/A-Verhalten besitzen. Eine funktionale Semantikdefinition reicht also für das Verständnis eines Programms nicht mehr aus, die Auswertungsstrategie ist mit in die Semantik hineinzukodieren. Über diesen Nachteil hinaus gleichen interaktive funktionale Programme sehr oft prozeduralen Programmen. Der Strichpunkt `;` wird so häufig als Kombinator von Continuations bzw. Monadenfunktionen verwendet, daß sich die Frage stellt, ob es nicht besser wäre, hier tatsächlich statt komplexer funktionaler Programmierung die wesentlich verständlichere imperative Programmierung zu verwenden. Statt eines funktionalen Verifikationskalküls bietet sich hier eine Verifikationsmethode mit Hoare-Regeln an.

Diese Überlegungen motivieren die Idee, E/A mittels imperativer Programmierung durchzuführen. Das funktionale Paradigma wird in einer ersten Schicht verwendet, um abstrakte Datentypen zu definieren und darauf Funktionen zur Verfügung zu stellen. Damit können komplexe Algorithmen auf diesen Datentypen auf abstrakte Weise formuliert werden. Eine darüberliegende zweite Schicht bietet imperative Programmierung mittels Zuweisungen, Aufrufen und Kontrollstrukturen, die die Programmierung expliziter Seiteneffekte, wie etwa die E/A erlauben. Der Erfolg graphischer Benutzeroberflächen ist ganz wesentlich auf die Benutzung des seiteneffektbehafteten, objektorientierten Paradigmas zurückzuführen. Deshalb wird in der zweiten Schicht das objektorientierte Paradigma verwendet. Dadurch werden die drei Konzepte der Objektorientierung *Kapselung*, *Identität* und *Vererbung* eingeführt. Das wesentlichste Konzept ist die Kapselung. Sie erlaubt es eine Gruppe von Attributen zu einer Einheit, dem *Objekt* zu gruppieren und nach außen unzugänglich zu machen. Ein Zugriff auf den Zustand eines Objekts kann nur über *Methoden*-Aufrufe erfolgen.

Das zweite wichtige Konzept ist die Identität, die jedem Objekt unabhängig

von seinem Zustand mit Hilfe eines (unveränderlichen) *Identifikators* zugeordnet wird. Dieser Objektidentifikator wird in der imperativen Objektorientierung genutzt, um explizit Seiteneffekte zu programmieren. Dies ist ein Umstand, der gerne vergessen wird, wenn das Objekt-Paradigma in funktionale Sprachen kodiert wird. Gerade E/A läßt sich durch die explizite Unterscheidung des Objekt-Raums und der Menge der Objektidentifikatoren aber hervorragend behandeln, da damit Seiteneffekt-Programmierung in kontrollierter Weise möglich wird.

Das Verhalten und die Zustandsräume von Objekten werden durch Klassen beschrieben. Vererbung zwischen Klassen erlaubt die Wiederverwendung von Code. Dies ist allerdings eher ein Mißbrauch, der auch durch Text-Expansion eines Präprozessors erledigt werden könnte. Vererbung etabliert aber auch eine Subklassenbeziehung zwischen Klassen, die eine Substituierbarkeit von Objekten der Superklasse durch Objekte aus Subklassen erlaubt. Diese Substituierbarkeit bedeutet zunächst, daß Objekte aus Subklassen mindestens die Methoden-Schnittstelle der Superklasse zur Verfügung stellen müssen. Dadurch wird durch statische Überprüfung verhindert, daß ein „message not understood" auftritt, wie das in `Smalltalk` häufig der Fall ist. Dieses Substitutionsprinzip für Objekte erlaubt z.B. die Verwendung von Design Patterns — einer sehr effektiven Programmiertechnik. Es ist statisch überprüfbar, weil die Menge der Objektidentifikatoren gemäß der Subklassenhierarchie (zwischen Klassen) in einer Subtyphierarchie strukturiert ist.

Es gibt mehrere Ansätze, z.B. [AC93], [CAR93] und [GRO95], funktionale und objektorientierte Paradigmen in einer Sprache zu vereinen. Die meisten dieser Ansätze kodieren das Objekt-Konzept innerhalb der funktionalen Welt. `GOS` (Kurzform für `Gofer Objekt-System`) versucht demgegenüber einen anderen Ansatz: Es schichtet beide Paradigmen auf zwei Ebenen übereinander und gibt so dem Programmierer die Möglichkeit zwischen diesen Ebenen nach Bedarf zu wechseln. Ein Vorteil dieser Schichtung ist, daß nur die Identifikatoren, die keine innere Struktur besitzen, nicht jedoch Objekte selbst funktional typisiert werden müssen. Dadurch wird das Typsystem deutlich einfacher.

Im folgenden werden die wesentlichsten Eigenschaften der Sprache `GOS` vorgestellt. Im letzten Abschnitt wird der derzeitige Stand der `GOS`-Implementierung beschrieben und einige Ausblicke gegeben.

2 Die Sprache `GOS`

Die Sprache `GOS` setzt sich zusammen aus zwei Schichten, einer imperativen objektorientierten Schicht und einer darunterliegenden funktionalen Schicht. Im folgenden werden zunächst beide erklärt und dann beschrieben, wie diese zusammenhängen.

Generell wird `GOS` durch folgende Eigenschaften charakterisiert:

- Objekte als Kapselungseinheiten für Attribute und Methoden,

- Sichtbarkeitsangaben für Attribute und Methoden,
- Objekte sind dynamisch erzeugbar,
- Klassen sind Beschreibungseinheiten für Objekte,
- Multiple Vererbung zwischen Klassen,
- Garbage Collection für Objekte,
- Strenge Typisierung,
- Mächtiges Konzept zur Datentypdefinition,
- Unterscheidung zwischen Objekten und Objektidentifikatoren und
- Dynamische Bindung beim Methodenaufruf.

2.1 Objektorientierte Schicht

Die objektorientierte Schicht ist inhaltlich stark an Programmiersprachen wie `C++` und `Eiffel` angelehnt, wenn sie auch deutliche syntaktische Unterschiede besitzt.

Ein Programm ist, wie in Abbildung 1 gezeigt, eine Sammlung von Klassendefinitionen und Definitionen global bekannter Objekte, die eine ähnliche Rolle wie globale Variable einnehmen, jedoch nicht durch Zuweisung verändert werden dürfen. Weitere Toplevel Definitionen werden im Abschnitt über die funktionale Schicht besprochen.

Klassen

Eine *Klasse* wird durch die Schlüsselworte `CLASS` und `ENDCLASS` gekennzeichnet. Als erstes steht eine (optionale) Auflistung der Superklassen (`SUBCLASSOF`). `GOS` erlaubt Mehrfachvererbung. Es darf in der Vererbungshierarchie jedoch kein Zyklus entstehen. Anschließend werden in beliebiger Reihenfolge, die *Member*, bestehend aus einem Konstruktor mit dem Namen `NEW`, neuen `Attributen`, sowie neuen und redefinierten `Methoden` angegeben. Attribut und Methodennamen sind kleingeschrieben, dasselbe gilt für Variablen- und Parameternamen. Im Gegensatz dazu beginnen Klassennamen mit einem Großbuchstaben.

Sichtbarkeitsangaben

Vor einer Attribut und Methodendefinition kann eine Sichtbarkeitsangabe der Form `PUBLIC`, `PROTECTED` oder `PRIVATE` stehen. Eine Sichtbarkeitsangabe bleibt gültig bis zur nächsten Angabe, anfangs gilt Sichtbarkeit `PUBLIC`. Der Konstruktor ist immer `PUBLIC`, Attribute immer `PROTECTED` oder `PRIVATE`.

Die Semantik dieser Angaben unterscheidet sich leicht von anderen Sprachen. Wir unterscheiden zwischen einer Sichtbarkeitseinschränkung im Quelltext

```
CLASS Cell
  PROTECTED v :: Int;                     -- Attributdefinition
  PUBLIC NEW (nv::Int) =                  -- Konstruktor
    BEGIN v := nv;                        -- Zuweisung
        io!print("Hallo Welt!\n");        -- Methodenaufruf
    END;
  get () :: Int =                         -- Methodendefinition
    BEGIN RETURN v;                       -- Ergebnisrueckgabe
    END;
ENDCLASS Cell;

CLASS ChangeableCell SUBCLASSOF Cell   -- Ableitung einer Klasse
  PRIVATE cl :: [Int];                    -- Liste von Integern
  PUBLIC NEW ...
  add (n:Int) =
    BEGIN cl := n:cl;                      -- Aenderung merken
          v := v+n;
    END;
  get_number_of_changes () :: Int =
    BEGIN RETURN length(cl); END;
ENDCLASS ChangeableCell

OBJECT io := IO!NEW();
OBJECT cc := ChangeableCell!NEW();
```

Abbildung 1: Teil eines Beispielprogramms

	Klassen-global	Klassen-lokal
Objekt-global	PUBLIC	-
Objekt-lokal	PROTECTED	PRIVATE

Abbildung 2: Kombination der Sichtbarkeitsangaben

- der Name des Members ist global bekannt (Klassen-global),
- er ist nur in der Klasse bekannt (Klassen-lokal)

und während der Laufzeit:

- das Member ist von allen Objekten zugreifbar (Objekt-global) und
- es ist nur vom eigenen Objekt aus zugreifbar (Objekt-lokal).

Diese Kriterien sind wie in Tabelle 2 beschrieben kombiniert. Während `PUBLIC` und `PRIVATE` selbsterklärend sind, bedarf `PROTECTED` einer Erläuterung: Member mit dieser Sichtbarkeit sind syntaktisch außerhalb der definierenden Klasse bekannt, können aber nur vom Objekt auf sich selbst angewendet werden. Deshalb ist eine Verwendung dieser Member nur in Subklassen möglich. Man beachte, daß anders als z.B. in `C++` `PRIVATE` und `PROTECTED` auf Objektebene wirken, also verschiedene Objekte derselben Klasse nicht gegenseitig auf ihren Zustand zugreifen können.

Die Sichtbarkeit eines Members kann in einer Subklasse von `PROTECTED` zu `PUBLIC` geändert werden. Private Member sind in Subklassen nicht mehr (direkt) zugänglich (aber trotzdem vorhanden); sogar deren Namen können neu verwendet werden. Die statische Typsicherheit wird gewährleistet, indem der Test auf Objekt-Lokalität durch eine syntaktische Einschränkung gesichert wird. Siehe dazu die Definition des Methodenaufrufs.

Attribute

Die Attribute eines Objekts sind Elemente funktionaler Datentypen. So ist `v` vom Typ `Int` und `cl` vom Typ `[Int]` (Liste über Integerzahlen). Bei Erzeugung eines Objekts sind diese Attribute undefiniert und müssen vor Verwendung erst besetzt werden.

Die Menge der Attribute eines Objekts der Klasse `K` bestimmt sich aus den in dieser Klasse angegebenen Attributen, sowie aller Attribute aus Superklassen, wobei nur die als `PROTECTED` deklarierten Attribute aus Superklassen zugänglich sind. Jedes Attribut wird nur einmal geerbt, Vererbung von Attributen mit gleichen Namen aber unterschiedlichem Typ wird als syntaktisch falsch zurückgewiesen.

Das spezielle Attribut `self::K` beinhaltet den eigenen Objektidentifikator. Es ist automatisch definiert, privat und kann nicht modifiziert werden.

Methoden

Methoden bilden die Schnittstelle eines Objekts. Sie können unter anderem genutzt werden, den Zustand eines Objekts zu erfragen und zu modifizieren. Eine Methode hat eine Liste von Parametern, die ähnlich wie Attribute funktional typisiert sind. Optional kann ein Ergebnistyp angegeben werden. In diesem Fall wird gefordert, daß eine entsprechende Ergebnis-Übergabeanweisung (`RETURN`) im Methodenrumpf steht.

Der Rumpf einer Methode beginnt eine mit `VAR` eingeleitete, optionale Definition lokaler Variable, die wiederum Elemente funktionaler Datentypen sind und nach dieser Vereinbarung erst initialisiert werden müssen. Dann folgt ein in `BEGIN` und `END` eingeschlossener Block von imperativen Anweisungen.

Methoden können bei Vererbung redefiniert werden, in diesem Fall kann der Rumpf einer Methode ersetzt werden, die Signatur bleibt jedoch erhalten. Erbt eine Klasse aus mehreren Superklassen unterschiedliche Methodenimplementierungen, so ist in der erbenden Klasse eine Redefinition der Methode vorzunehmen, um damit Mehrdeutigkeiten aufzulösen. Dies ist nicht notwendig, wenn durch eine Diamant-artige Vererbungsstruktur dieselbe Methodenimplementierung über verschiedene Wege geerbt wird.

Konstruktor

Der Konstruktor wird ähnlich definiert wie eine Methode. Er besitzt jedoch keinen expliziten Ergebnistyp und im Rumpf keine `RETURN`-Anweisung. Der Konstruktor wird anders als Methoden gestartet, genau nachdem ein neues Objekt dieser Klasse erzeugt wurde. Er dient der Initialisierung des Objektzustands. Konstruktoren aus Superklassen werden dabei im Unterschied zu `C++` nicht aufgerufen.

Anweisungen

Es gibt folgende Arten von Anweisungen:

Zuweisung `var := Ausdruck` mit einem Attribut oder einer lokalen Variable `var` und einem funktionalen `Ausdruck`.

Methodenaufruf `oid!methode(Argumente)` mit einem funktionalen Ausdruck `oid`, der das Zielobjekt des Methodenaufrufs identifiziert, einem Methodennamen und einer möglicherweise leeren Liste von Argumentausdrücken. Jedes Argument muß vom in der Methodendefinition angegebenen Typ sein. Ebenso muß `oid` einen Objektidentifikator-Typ besitzen, der sichert, daß das damit identifizierte Objekt den Methodenaufruf versteht. Damit wird strenge Typsicherheit gewährleistet.

Wird ein Methodenaufruf an das eigene Objekt gesendet, so kann dies mit der Abkürzung `!methode(Argumente)` geschehen. Private Methoden können nur vom eigenen Objekt aufgerufen werden. Deshalb muß beim Aufruf solcher Methoden diese Kurzform gewählt werden, oder für `oid` syntaktisch die Variable `self` stehen.

Methodenaufrufe werden dynamisch gebunden, das heißt erst zur Laufzeit wird die tatsächlich aufgerufene Methode abhängig vom empfangenden Objekt ausgewählt. Ausnahmen sind private Methoden, wo eine statische Vorauswahl stattfinden kann, und Aufrufe von Methoden aus Superklassen, die folgende Syntax haben: `!Klasse!methode(Argument)`. Redefinierte Methoden aus Superklassen bleiben damit in Subklassen zugänglich.

Dem Methodenaufruf kann eine Zuweisung vorangestellt werden um das Ergebnis eines Aufrufs abzulegen `var:=oid!method(...)`.

Konstruktoraufruf `var:=Klasse!NEW(Argumente)` ist dem Methodenaufruf syntaktisch ähnlich. Weil Klassen keine zur Laufzeit des Systems existierenden Einheiten sind, sollten Methodenaufruf und Konstruktoraufruf nicht verwechselt werden. Der Konstruktoraufruf erzeugt zunächst ein Objekt der angegebenen Klasse und startet dann die in `NEW` angegebene Initialisierung. Als Ergebnis des Konstruktoraufrufs wird ein Identifikator des erzeugten Objekts übergeben, der in einer Variable abgespeichert werden kann (aber nicht muß).

Ergebnisübergabe `RETURN Ausdruck` beendet die Methode und übergibt das Ergebnis. Hat die Methode keinen Ergebnistyp, so kann `RETURN` verwendet werden, um diese zu beenden.

While-Schleife `WHILE b DO kdos ENDDO` bietet eine Schleife, die solange die Kommandosequenz `kdos` ausführt, wie der Boolesche Ausdruck `b` den Wert `True` liefert. Minimale Durchlaufzahl ist 0.

For-Schleifen gibt es in zwei Varianten. Eine Variante durchläuft einen Integerbereich und ist von der Form `FOR n := start TO ende DO kdos ENDDO`. Die Variable `n` nimmt dabei in aufsteigender Reihenfolge die Werte der Ausdrücke `start` bis `ende` an und durchläuft damit die Schleife. Beide Ausdrücke werden nur zu Beginn einmal ausgewertet. Die Schleife wird mindestens 0 mal durchlaufen. Der Wert von `n` ist nach Ende der Schleife nicht festgelegt.

Die zweite Variante der For-Schleife wurde für den häufig benutzten, und in `GOS` bereits vordefinierten Datentyp Listen eingeführt. Die Syntax lautet `FOR l IN liste DO kdos ENDDO`. Dabei wird der Ausdruck vom Typ `[a]` (Liste über einen Ausdruck des Typs `a`) zu Beginn ausgewertet und die Variable `l` mit dem Typ `a` durchläuft diese Liste elementweise. Der Wert

der Variablen `l` nach Ende der Schleife ist nicht festgelegt. Für eine leere Liste wird die Schleife 0 mal durchlaufen.

Zu Beachten: Aufgrund der lazy-Auswertungsstrategie von Listen ist es möglich, For-Schleifen über unendliche Listen zu iterieren. In diesem Fall muß ein Abbruch der Auswertung mit `RETURN` im Rumpf der Schleife erfolgen.

Fallunterscheidung `IF b THEN` $\texttt{kdos}_t$ `ELSE` $\texttt{kdos}_f$ `ENDIF` erlaubt die bedingte Ausführung von Kommandos, abhängig von der Booleschen Bedingung `b`. Der Else-Teil ist optional.

Eine spezielle Form der Fallunterscheidung wird verwendet um den dynamischen Typ von Werten in Variablen zu ermitteln und ein Retract (Down-Cast) durchzuführen. Die Syntax lautet `TYPEIF var IS T THEN ...`. Beinhaltet die Variable `var` einen Wert vom Typ `T`, so wird in den Then-Teil verzweigt, sonst in den optionalen Else-Teil. Im Then-Teil hat die Variable `var` den Typ `T`. Damit der Then-Teil überhaupt erreichbar ist, muß der Typ `T` ein Subtyp des Typs der Variable `var` sein.

Objektidentifikatoren

In `GOS` wird zwischen Objekten und Objektidentifikatoren unterschieden. Objekte sind *klassifiziert*, Objektidentifikatoren sind *getypt*. Objekte sind keine Elemente funktionaler Datentypen, sondern Elemente von Klassen. Objekte können daher nicht als Ergebnisse von Ausdrücken entstehen, sie können auch nicht als Argumente an Methodenaufrufen übergeben werden.

Stattdessen wird der Objektidentifikator als Referenz auf ein Objekt übergeben. Objektidentifikatoren sind Elemente funktionaler Datentypen und können daher als Ergebnisse von Ausdrücken, als Argumente u.ä. mehr auftreten.

Die Menge der Objektidentifikatoren ist ähnlich strukturiert wie die Klassen. Zu jeder Klasse `K` existiert ein gleichnamiger Datentyp `K`, der eine (unendliche) Menge von Identifikatoren beinhaltet. Zusätzlich dazu gibt es die beiden Identifikatortypen `NIL` und `OID`, denen keine Klassen entsprechen. Erstere enthält nur den speziellen Identifikator `Nil`, letztere gilt als Zusammenfassung aller Objektidentifikatoren in einer Sorte. Die Vererbungshierarchie der Klassen induziert eine Subtypinghierarchie auf der Menge der Objektidentifikatoren, wie in Abbildung 3 gezeigt.

Auf das modifizierte Typsystem, das durch die Verwendung von in `Gofer` nicht vorhandenem Subtyping entsteht, wird später genauer eingegangen.

Global bekannte Objekte

Neben der Klassendefinition ist die Definition von global bekannten Objekten auf oberster Ebene erlaubt. Definiert wird ein global bekanntes Objekt wie folgt:

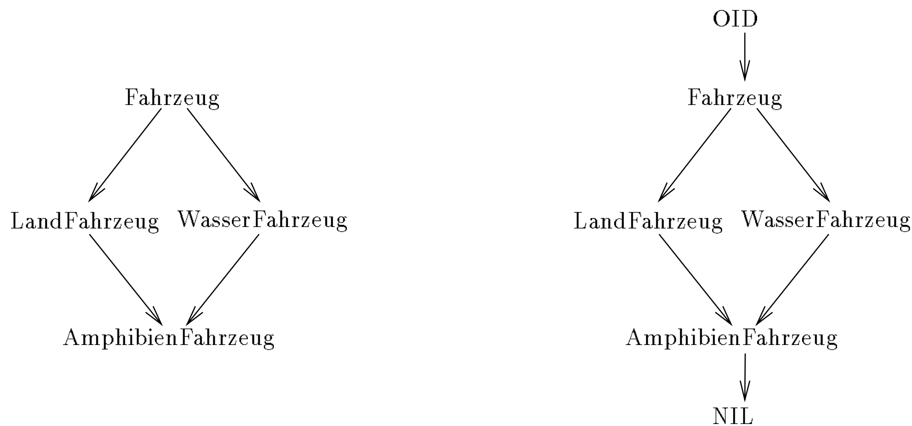

Abbildung 3: Subklassen- und Subtyphierarchie

`OBJECT o := Klasse!NEW(Argumente)`. Dabei wird eine Konstante `o::Klasse` eingeführt. Diese entspricht einem Objektidentifikator, der das so erzeugte Objekt bezeichnet.

Global bekannte Objekte besitzen drei Aufgaben. Zuerst spielt ein global bekanntes Objekt die Rolle einer globalen Variable, die zu Beginn mit dem Identifikator eines Objekts besetzt wird, und die nicht mehr modifiziert werden darf. Es können jedoch Methodenaufrufe an das Objekt gesendet werden, da der mit einer Objektdefinition vereinbarte Name wie eine Konstante mit Objektidentifikator-Typ verwendet werden kann. Damit werden globale Variable ersetzt, die in `GOS` nicht vorhanden sind. Vorteil dieser Technik gegenüber globalen Variablen ist, daß keine Kommunikation über gemeinsamen Speicherplatz stattfindet, sondern auch hier Kapselung in einem Objekt vorliegt.

Zum zweiten können damit wichtige Objekte, wie das Ein-/Ausgabeobjekt `io` definiert werden, die von jedem Objekt aus zugänglich sein sollten. Zum dritten wird damit das Hauptprogramm festgelegt. Denn die in der Quelldatei verteilten Definitionen globaler Objekte werden in der Reihenfolge ihres Auftretens abgearbeitet. Ist das letzte globale Objekt abgearbeitet, so ist die Ausführung des Programms beendet. Typischerweise sollte also der Konstruktor `NEW` des letzten definierten Objekts das Hauptprogramm enthalten.

Zu beachten ist, daß die textuelle Reihenfolge global bekannter Objekte relevant ist. Wird etwa auf ein nocht nicht initialisiertes global bekanntes Objekt zugegriffen, so entsteht ein Laufzeitfehler.

2.2 Funktionale Schicht

`GOS` Programme bilden eine Obermenge der `Gofer` Programme. Insbesondere sind alle funktionalen Elemente der Sprache `GOS` aus Gofer übernommen und im dortigen Manual ausführlich erklärt.

Sogar `Gofer` Namenskonventionen werden übernommen. Attribute, Variablen und Parameter beginnen mit Kleinbuchstaben, Klassennamen mit Großbuchstaben. `Gofer` Deklarationen können wie bisher als Top-Level Deklarationen definiert werden. Zu beachten ist nur, daß bestimmte Worte der `GOS` Grammatik als Schlüsselworte reserviert sind. Beispielsweise wird das Ausrufezeichen `!` zum Methodenaufruf verwendet.

Ausdrücke

Auf rechten Seiten von Zuweisungen, als Bedingungen in Kontrollstrukturen, als Anfang und Ende, bzw. Listen von For-Schleifen, als Argumente von Methoden- und Konstruktoraufrufen, sowie zur Bezeichnung des Zielobjekts eines Methodenaufrufs stehen *funktionale Ausdrücke*. Dies sind reine `Gofer` Ausdrücke, wie sie im dortigen Manual beschrieben sind. Ein funktionaler Ausdruck wird seiteneffektfrei ausgewertet, er kann also selbst keinen Methodenaufruf beinhalten. Ausdrücke sind getypt. So können Bedingungen nur Boolesche Ausdrücke sein und der Ausdruck, der das Zielobjekt eines Methodenaufrufs beschreibt, muß einen passenden Objektidentifikator-Typ haben.

Innerhalb eines Ausdrucks können Attribute, lokale Variable und Parameter wie Konstante verwendet werden. Die Auswertungsstrategie für Ausdrücke ist lazy. Das heißt ein Ausdruck kann durchaus einen undefinierten Wert bezeichnen, oder wie etwa bei Listen unendlich sein. Eine Auswertung findet erst bei Benutzung des Ausdrucks statt. Zu beachten ist, daß im Gegensatz dazu die imperative Teilsprache der Anweisungen sofort ausgeführt wird.

Datentypen

Neben Ausdrücken werden in `GOS` funktionale Datentypen verwendet. Bei der Definition von lokalen Variablen, Attributen und Parametern werden deren Datentypen mitangegeben. Datentypen sind monomorphe Typausdrücke. Das heißt, sie dürfen beispielsweise aus Listen, Tupel und Funktionskonstruktor aufgebaut sein, jedoch keine Typvariable enthalten.

Vordefinierte Datentypen

In `GOS` stehen dieselben vordefinierten Datentypen wie in `Gofer` zur Verfügung. Dies sind die Datentypen

- `Bool` mit den Booleschen Werten `True` und `False` und den üblichen Verknüpfungen,

- `Int` mit allen Integerzahlen, den Konstanten `1`, `-42` etc. und den üblichen Rechenoperationen,
- `Float`, die Fließkommazahlen, ebenfalls mit Konstanten und Rechenoperationen,
- `[a]`, der polymorphe Typ über Listen mit entsprechenden Listenoperationen,
- `(a,b)`, der Möglichkeit beliebigstellige Tupel zu bilden und
- `(a->b)`, der Funktionstyp, dessen Elemente Funktionen sind.

In `GOS` ist es erlaubt, Attribute von Funktionstyp zu definieren. Diese sind allerdings nicht zu verwechseln mit Methoden. Erstere können nicht auf Objekten arbeiten, da Objekte keine Elemente funktionaler Datentypen sind.

Datentyp- und Funktionsdefinitionen

`Gofer` stellt zur Definition neuer Datentypen das `data`-Konstrukt zur Verfügung. Für die Definition von Funktionen (dies sind nichts anderes als Konstanten von Funktionstyp) kann Patternmatching verwendet werden. Genaueres siehe bei `Gofer`.

Objektidentifikator-Typen

Eine spezielle Art von Typen bilden die Objektidentfikator-Typen. Wie in Abbildung 3 gezeigt, sind diese in einer Subtyping-Hierarchie angeordnet. Dies ist gleichzeitig die einzige Form, Subtyping zwischen funktionalen Datentypen in `GOS` einzuführen.

Zu einer Klasse `K` wird ein gleichnamiger Typ `K` eingeführt, der Objektidentifikatoren enthält. Zu diesem Typ existiert kein Konstruktor; stattdessen werden Elemente dieses Typs gemeinsam mit den dadurch bezeichneten Objekten kreiert. Dadurch wird referentielle Integrität gewahrt. Für die Sorte `K` stehen nur die Vergleiche `==` und `/=`, sowie die lineare Ordnung `<` (`<=`, ...) zur Verfügung. Ein Objektidentifikator-Typ ist Instanz der Typklasse `ORD`.

Subtyp-Hierarchie

Die durch Vererbung zwischen Klassen definierte Hierarchie induziert eine Subtyp-Hierarchie zwischen den Objektidentifikator-Typen, wobei der oberste Typ `OID` und der unterste Typ `NIL` hinzugenommen werden. Dies sind die einzigen Subtyping-Beziehungen zwischen Grundsorten, der Programmierer kann keine weiteren etablieren.

Das Substitutionsprinzip für Objekte wird ermöglicht, indem ein Identifikator für ein Objekt einer Subklasse durch Coercement (Up-Cast) zu einem Element des Supertyps gemacht werden kann. Dazu dient eine polymorphe Funktion `coerce::a->b`, die für jedes Paar aus Subtyp `a` und Supertyp `b` zur Verfügung steht. Der Typinferenz-Algorithmus von `Gofer` wurde an dieser Stelle so modifiziert, daß Coercement auch implizit stattfindet, indem die `coerce`-Funktion wo notwendig implizit eingefügt wird.

Eine Umkehrung des Coercements, ein Retract (Down-Cast) existiert in der funktionalen Schicht nicht, dazu kann das `TYPEIF`-Konstrukt verwendet werden.

Die Subtyping-Relation induziert sich über Typkonstruktoren, wie Tupel und Listen, auf monotone Weise. Stehen etwa die Komponenten eines Tupeltyps paarweise in Subtyping-Relation, so auch der Tupeltyp. Der Funktionsraumkonstruktor ist jedoch, wie bei Subtyping-Hierarchien üblich, im ersten Argument antiton.

Weitere funktionale Deklarationen

Von `Gofer` werden die Möglichkeiten angeboten, Typklassen zu definieren und Instanzen daraus zu bilden. Diese Toplevel-Deklarationen stehen weiter ungehindert zur Verfügung.

Tatsächlich wird das Subtyping der Objektidentifikator-Typen intern durch eine zweistellige Typklasse realisiert.

3 Implementierung und Ausblick

Das `GOS`-Projekt wurde im Sommer 1994 an der TU München unter starker Studentenbeteiligung gestartet. Es ist seitdem eine auf den `Gofer`-Interpreter von Mark P. Jones beruhende Implementierung eines Interpreters für `GOS` geschaffen worden. Dieser besitzt eine Anbindung des Betriebssystems und des Graphik-Toolkits `Tk/Tcl` in Form zweier Klassenbibliotheken.

Einige Werkzeuge, unter anderem `GOS`-Quelltext erzeugende Versionen von `lex` und `yacc` stehen zur Verfügung. Ein Debugger und weitere Werkzeuge, z.B. zur Visualisierung von `GOS`-Quellcode, sind in Arbeit.

Kleinere Modifikationen der Sprache im Kleinen und eine Erweiterung um ein Modulkonzept ähnlich zu Oberon-2 sind angedacht, bzw. ebenfalls in Arbeit.

Nach nunmehr einjähriger Entwicklungszeit und einigen Erfahrungen in der Anwendung von `GOS` hat sich gezeigt, daß diese Sprache die in sie gesetzten Erwartungen erfüllen kann. In der Lehre wird heute in zunehmender Tendenz `Gofer` als Vertreter funktionaler Sprachen gelehrt, danach bietet sich häufig Modula-2 oder Pascal als prozedurale Sprache an. Zur Vermittlung objektorientierter Konzepte ist keine der heute zur Verfügung stehenden Sprachen als sehr geeignet anzusehen. `GOS` bietet sich als Erweiterung von `Gofer` geradezu an, diese Lücke zu schließen.

GOS ist eine Programmiersprache, mit der sich aufgrund der mächtigen Technik zur Datentypdefinition kleinere und mittlere Software sehr schnell und effektiv schreiben läßt. Mit Hilfe eines mächtigen Konzepts zur Definition von Datentypen werden wesentlich weniger Objekte als bisher für Programme vergleichbarer Größe benötigt. Die Objektlandschaft wird dadurch überschaubarer und leichter verständlich. Sie kann damit leichter gewartet und modifiziert werden.

Danksagung

Prof. Dr. Manfred Broy danke ich für die wohlwollende Unterstützung bei der Entwicklung der Sprache GOS. Für interessante Diskussionen und den Entwurf des GOS-Logos bedanke ich mich bei meinem Kollegen Franz Huber. Für die Unterstützung bei der Implementierung des Interpreters der Sprache GOS bedanke ich mich herzlich bei meinen Studenten, von denen ich hier stellvertretend Ingolf Krüger, Ulf Schünemann und Marc Sihling nennen möchte. Mein ganz besonderer Dank gilt Sonja Fuchslechner für ihre Hilfe und Geduld bei der Entwicklung von GOS.

Literatur

[AC93] R. M. Amadio and L. Cardelli. Subtyping Recursive Types. SRC Research Report 62, Digital Equipment Corporation, January 1993.

[CAR93] L. Cardelli. Extensible Records in a Pure Calculus of Subtyping. SRC Research Report 81, Digital Equipement Corporation, January 1993.

[GRO95] R. Grosu. *A Formal Foundation for Concurent Object Oriented Programming.* PhD thesis, Technische Universität München, Januar 1995.

[JON93] M. P. Jones. *An Introduction to Gofer*, 1993.

[MEY92] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[PAU91] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[STR91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.