Programmiersprachen

Vorlesung 3

Zusammenfassung

Georg Ernst Moser

Structuring the Data

Why Use (Static, Strong) Types?

Welche Typen meinen wir? Statisch, Stark, Dynamisch?

- **Verstecken der Maschinen-Repräsentation**: Es muss nicht auf irgendwelche Bitmuster zugegriffen werden, man braucht diese auch gar nicht zu kennen. Stattdessen wird auf abstrakte Datentypen (z.B. String,..) zugegriffen.
  Ziele: Es kann ein schönerer Programmierstiel und höhere Wartbarkeit erreicht werden.
- **Möglichkeit der Überprüfung durch den Compiler ob Typen konsistent verwendet werden:** Passen Operatoren und Operanden zusammen?
  Ziele: Lesbarkeit (durch Verwendungsvorgabe von Typen,...) , Korrektheit
- **Platzbedarf der einzelnen Typen bekannt**: Compiler kann statisch Platz für Typen reservieren.
  Ziele: Performance (weniger Zeiger, weniger Indirektionen), Speichereffizienz
- **Überladen**: Wie kann man überladene Routinen unterscheiden? → Typen angeben
  Ziele: Lesbarkeit, Performance (es kann bereits zu Compile Zeit unterschieden werden)

Elementare Typen

Elementare Typen sind Typen, welche nicht weiter zerlegt werden können (Grundeinheiten). Diese existieren in allen Sprachen und werden benötigt um „Zugang zur Maschine“ schaffen zu können, Speicherzellen anzusprechen.
Elementare Typen sind zu unterscheiden von **vordefinierten Typen**. Ein Beispiel für einen vordefinierten Typ, welcher nicht elementar ist, wäre „String“ in Ada: Dieser ist abgebildet als ein (dynamisches) Char-Array mit positiven Grenzen.
Es gibt auch elementare Typen, die nicht vordefiniert sind. Zum Beispiel Aufzählungstypen (Enum, type,..)

Aggregates and Type Constructors

Ein Aggregat ist die Zusammenfassung von elementaren Werten (Datenobjekten).
Typische Beispiele für Aggregate sind Arrays und Records. Diese hatten ursprünglich keinen Typ (Cobol) und wurden direkt erstellt.
Neuere Sprachen haben Typen für Aggregate. Man erstellt zuerst den Typ und denkt noch gar nicht über dessen Instanzen nach. Es werden Konstruktoren zum Erstellen von diesen Instanzen benötigt, welche aber meist von der Programmiersprache versteckt werden.
Datenkonstruktoren sind in fast allen Sprachen vorhanden (Nicht nur in objektorientierten, auch funktional,...)
Wichtige Vertreter von Aggregaten werden im Folgenden behandelt:

Cartesian Product

Das Kartesische Produkt ist Programmieren als Teil von Programmiersprachen zwar oft nicht unter diesem Namen bekannt, entspricht aber im Wesentlichen Structs, Records, Array..
„Das kartesische Produkt zweier Mengen ist die Menge aller geordneten Paare von Elementen der beiden Mengen, wobei die erste Komponente ein Element der ersten Menge und die zweite Komponente ein Element der zweiten Menge ist.“ - Wikipedia
Statt der Indizes werden in einem Record Namen verwendet.
In einem Array werden tatsächlich Indizes verwendet, weshalb es der eigentlichen Kreuzmenge sehr ähnlich ist.

Beispiel für ein struct in C++

```
typedef struct {
int no_of_edges;
float edge_size;
} reg_polygon;
reg_polygon a_pol = { 3, 3.14 };
```

Auf Felder wird mit der dot-Notation zugegriffen:

```
a_pol.no_of_edges = 4;
```

Finite Mapping

Endliche Abbildungen (Funktionen) bilden eine Domain in einen Range ab:

f: domain → range *Beispiel: f: integer → real*

Eine Endliche Abbildung ist eine Funktion mit endlicher Domain.

Es gibt **2 Möglichkeiten** eine Funktion zu definieren:

- **intentional**: Es wird die Intention angegeben wie ein Wert aus der domain in den range abgebildet wird. f(x) = x*2
- **extensional**: Es wird die Menge aller möglichen Paare angegeben, wobei diese aus jeweils einem Wert aus der Domain und einem Wert aus dem Range gebildet werden. In Programmiersprachen entspricht dies einem Array.

Array Types (Examples)

Beispiel in C:

```
char digits[10];
for (i=0; i<10; i++) {
      digits[i] = ' ';
}
```

Es wird nur die Größe des Arrays angegeben. Die Indizes sind vorgegeben und liegen im Intervall [0,n-1]. In C werden keine Standardwerte für die Elemente eines Arrays vergeben, weswegen es explizit initialisiert werden muss. Beim Zugriff auf das i-te Element eines Arrays kann es schwierig sein sicherzustellen, dass dieses wirklich existiert (ist Parameter i korrekt? Zum Beispiel von stdin eingelesen).
Erfolgt ein Zugriff auf ein Element außerhalb der definierten Grenzen, tritt ein Fehler auf. Moderne Compiler können relativ genau überprüfen ob nur auf Werte innerhalb des Arrays zugegriffen wird. Da man sich aber nicht auf den Compiler verlassen möchten (jeder Compiler arbeitet anders) wird dies dynamisch, oder gar nicht überprüft.

Beispiel in Pascal:

```
var x: array[2..5] of integer;
type manufacturer = (ibm, dec, hp, sun);
type m_data = array[manufacturer] of integer;
var m_profits, m_empl: m_data;
```

Hier wird der Indexbereich angegeben, in diesem Fall von 2 bis 5, was unter Umständen leichter Verständliche Paarbildungen ermöglicht.
Es können aber auch Aufzählungstypen als Indizes verwendet werden. Hierbei ist es nicht mehr möglich auf Elemente außerhalb des Arrays zuzugreifen. In diesem Fall kann zum Beispiel kein Index vom typ manufacturer angegeben werden, der auf eine Speicherzelle außerhalb des Arrays führt.

Direktes Initialisieren von Arrays

C :
```
char digits[5] = {'a','b','c','d','e'};
```

Ada:
```
X: array(INTEGER range 2..6) of INTEGER := (0,2,0,5,-33);
```

In Ada kann angegeben werden welchen Typ der Range hat und dieser eingeschränkt werden. Ist der Index vom Typ Integer kann nicht mit einem Short Integer zugegriffen werden (in Ada nicht kompatibel).
In Ada gibt es eine Vielzahl an Möglichkeiten die Elemente welche ein Array enthalten soll zu beschreiben.

Mehrdimensionale Arrays:

C:
```
int y[10][20];
```
In C entspricht ein mehrdimensionales Array einem Array aus Arrays.

Ada:
```
Y: array(1..27, M..N) of INTEGER;
```
In Ada wird dies als echt 2-dimensionales Array realisiert.

Man kann bei Arrays bezüglich deren Implementierung zwischen echten und unechten unterscheiden:
echte Arrays: Zum Beispiel in C. Zeigen direkt auf den Speicher, bzw. das erste Element des Arrays.
unechte Arrays: zum Beispiel in Java. Hier gibt es eine Indirektion mehr, da zuerst ein Zeiger auf das Array zeigt und von diesem dann auf die einzelnen Elemente zugegriffen wird.
In Java können im Array ganze Zahlen stehen. Es gibt aber auch Sprachen, welche in einem Array wiederum nur Zeiger auf die eigentlichen Elemente speichert und somit eine weitere Indirektion hinzukommt.

In diesem Sinne unterscheidet sich auch der Zugriff auf Mehrdimensionale Arrays zum Beispiel in C und Java.
C: Aus 2 Indizes wird ein Indizes berechnet, auf welchen dann zugegriffen werden kann. Die erste Dimension enthält die Anfangs-Adressen der Arrays der 2ten Dimension.
Java: Die erste Dimension ist ein Array, das Zeiger auf Arrays enthält. Aus diesem Grund ist es auch möglich „dreieckige“ Arrays aufzubauen: Die Arrays 2ter Dimension sind verschieden lang. In C wäre das nicht möglich (bei mehrdimensionalen Arrays)

Array Slicing in Ada

Jeder Ausschnitt aus einem Array ist ein Slice. Mit diesen kann genauso gearbeitet werden wie mit einem „normalen“ Array. Dies könnte seinen Ursprung darin haben, dass in Ada Arrays zur String-Repräsentation verwendet werden (einfache String Operationen durch Slices möglich).

Es können zum Beispiel Teilbereiche eines Arrays an ein anderes Array zugewiesen werden.

```
X(2..5) := X(3..6);
```

Associative Data Structures

Bei Assoziativen Datenstrukturen können beliebige Typen als Indizes verwendet werden.

Beispiel in SNOBOL4:

```
T = TABLE()             //Erzeuge neues Array
T<’RED’> = ’WAR’        //T an der Stelle 'Red' (String) soll 'WAR' enthalten
T<6> = 25               //T an der Stelle 6 (Int) soll 25 enthalten
T<4.6> = ’PEACE’        //T an der Stelle 4.6 (float) soll 'PEACE' enthalten
```

Domain und Range sind hierbei unendlich groß.
Ein Vorteil ist, dass nie mit einem falschen Index zugegriffen werden kann. Ein ungültiger Index liefert etwas wie „null“ zurück.
Assoziativen Datenstrukturen können auch als eine Form von Hash Maps gesehen werden.

Ein Problem der Assoziativen Datenstrukturen kann am Beispiel von T<4.6> beschrieben werden. 4.6 ist eine Fließkommazahl. Was passiert aber wenn der Aufruf durch Rundung folgendermaßen erfolgt T<4.599999999>?

Assoziativen Datenstrukturen werden vor allem in dynamischen Sprachen verwendet da nur wenig Angaben über den statischen Typ in ihnen gemacht werden kann.

Union

Definition einer Union in C:

```
union address {
short int offset;
long unsigned int absolute;
};
```

Eine Union ermöglicht es, Variablen zu definieren, die sich denselben Speicherbereich teilen. Der Speicherplatzbedarf einer Union wird bestimmt durch den Speicherplatzbedarf seiner "größten" Komponente. Sie darf aber immer nur einen Wert enthalten, also in obigem Beispiel entweder *offset* oder *absolute*.
Das Problem dabei ist, dass nicht bekannt ist welchen Wert die Union gerade hält (C spezifisches Problem).

Aus diesem Grund werden Unions meistens mit einem Deskriptor (als enum realisiert) zusammengeführt (eine Struktur darüber gebaut), welcher Aufschluss über den momentan enthalten Typ gibt.

```
enum descriptor {abs, rel};
typedef struct {
address location;
descriptor kind;
} safe_address;
```

Discriminated Union

Der **variant record** in Pascal ist ein gemeinsames Konstrukt für union und struct.

```
natural = 0..maxint;
address_type = (absolute, offset);
safe_address =
      record
            case kind: address_type of
                  absolute: (abs_addr: natural);
                  offset: (off_addr: integer)
      end
```

Die Felder (*absolute* und *offset*) des records (ähnlich einem struct in C) können von einem bestimmten Typ (in diesem Beispiel *kind*) abhängig gemacht werden.
Es kann nur auf *absolute* oder *offset* zugegriffen werden, wenn *kind* den entsprechenden Wert beinhaltet.
Beispiel: Wenn *address_type* den Typ *absolute* hat, enthält der record das Feld *abs_addr*.

In Ada wird ein modernerer Ansatz verwendet bei dem der Deskriptorwert (in diesem Beispiel *kind*) nicht dynamisch geändert werden kann. Er wird beim Erstellen fixiert und bleibt immer gleich, weshalb auch eine einfachere statische Überprüfung möglich ist.

Powerset

Ein Powerset ist ein Begriff der ebenfalls aus der Mathematik kommt und steht für die Menge aller Teilmengen einer Menge.
Bsp: M ={1,2,3} Teilmengen: {],{1},{2},{3},{1,3},{1,2},{2,3},{1,2,3}
In Programmiersprachen entspricht die Menge einem Typ. Ein Powerset ist demnach ein Typ von Sets.

Beispiel in Pascal (Sprache aus der auch Powersets kommen):

```
type option = (list, optimize, save, exec); //Menge
option_set = set of option; //Menge aller options
var active_options: option_set;
...
active_options := [optimize, save]; //Teilmenge
if exec in active_options then …
```

Im Wesentlichen entsprechen Powersets allerdings einer benutzerfreundlicheren Form eines Bitsets. Powersets sind meist genau als ein Maschinenwort darstellbar: Jedes einzelne Bit in diesem Wort entspricht einem Wert. Wenn Bit 1 gesetzt ist, ist *list* in der Menge, wenn nicht, dann nicht. Wesentlich mehr können auch Powersets nicht. Aufgrund dieser einfachen Darstellung kann sehr effizient mit ihnen gerechnet werden.

Recursive Data Types

Heute sind fast alle Datentypen in irgendeiner Form rekursiv; eine Entwicklung die relativ lange gedauert hat.

Formale Definition von rekursiven Datentypen:

```
bin tree = {nil} ∪ (integer × bin tree × bin tree)
int list = {nil} ∪ (integer × int list)
```

Man könnte annehmen eine derartige Typdefinition steht für unendliche Datentypen. Die Annahme ist jedoch nicht richtig, da die Typdefinition eigentlich wie folgt zu interpretieren ist:
Am Ende steht immer ein „nil“ und davor endlich viele Elemente. Als mathematische Erklärung dafür dienen dies sogenannten „rational trees“. Alles was mit rationellen Bäumen dargestellt werden kann, kann auch als rekursiver Typ dargestellt werden.

Beispiel in ML[1]

```
fun find(el, nil) = false                 //Wenn Liste leer
|     find(el, x::xs) =                   //x → erstes Element;xs → Rest
            if el = x then true           //wenn gesuchtes Element = x
                  else find(el, xs) ;     //rekursiver Aufruf mit Rest
```

Hier kann deutlich gesehen werden, dass immer auf ein Element nach dem anderen zugegriffen wird und über Pattern matching 2 Fälle unterschieden werden. Dies ist genau die Art wie mit rekursiven Typen gearbeitet wird.

Beispiel in C:

```
typedef struct list {
      int val;
      struct list* next;
} int_list;
int_list* head;
```

In einer Struktur befindet sich ein Element (ein Zeiger), welches bereits auf eine derartige Struktur zeigt. Es können allerdings nur Namen verwendet werden, welche bereits definiert wurden. Innerhalb des *typedef* ist der Name des struct *list* eigentlich noch nicht definiert, weswegen ein Pointer verwendet werden muss.

Beispiel in Ada:

```
type INT_LIST
type INT_LIST_REF is access INT_LIST;
type INT_LIST is
      record
            VAL : INTEGER;
            NEXT: INT_LIST_REF;
      end;
HEAD: INT_LIST_REF;
```

In Ada wird Deklaration und Definition klar getrennt. Am Anfang wird *INT_LIST* nur deklariert. Anschließend wird ein Zeiger darauf (*INT_LIST_REF*) definiert. Erst dann kann *INT_LIST* definiert werden, weil *INT_LIST_REF* bereits darinnen vorkommt.

1 statisch typisierte funktionale Sprache, ähnlich Haskell, aber strict statt lazy. Der Begriff Typinferenz wurde erstmals in ML verwendet.

Unsafety of Pointers

Die Verwendung von Zeigern bringt viele Unsicherheiten.

- **Untypisierte Zeiger**: Erst in der Verwendung wird angenommen worauf der Zeiger zeigt. Man arbeitet mit dem Zeiger als ob er zum Beispiel auf einen char zeigen würde. Sicherheit gibt es hierbei jedoch keine, der Typ gibt keinen Aufschluss darüber.
  Lösung: Typisierte Zeiger.
  Dies gilt auch für dynamische Sprachen: Bei diesen ist der Typ als Teil der Referenz gespeichert.

- **Zeigerarithmetik**: Durch das Umbiegen von Zeigern können diese leicht auf eine ungültige Adresse zeigen.
  Lösung: Verbieten von Zeigerarithmetik (auf Kosten der Effizienz)

- **Hängende Zeiger durch Verletzen von Scopes**
  Bsp: Eine lokale Variable wurde im Stackframe angelegt (zum Beispiel als Ergebnis einer Berechnung welche in einer Routine durchgeführt wurde) und ein Zeiger darauf wird zurückgegeben. Dieser zeigt jedoch in den Stackframe der bereits abgeschlossenen Routine, welcher bereits abgebaut wurde.
  Lösungen:
  - Verwendung des Adressoperators verbieten
  - alles auf den Heap schreiben (nichts auf den Stack)
  - Laufzeitüberprüfungen (erkennt Probleme im Zusammenhang mit dem Adressoperator): Diese Überprüfungen sind jedoch extrem teuer.

- **Hängende Zeiger aufgrund von Speicherfreigabe**:
  Lösung: Garbage Collection statt expliziter Speicherfreigabe.
  Garbage Collection hatte lange den Ruf ineffizient zu sein. Mittlerweile ist bekannt, dass GC sehr effizient im Vergleich zu „handgestrickten“ Lösungen ist. Alle jüngeren Sprachen verwenden GC, außer wenn sie im Bereich von Echtzeitsystemen zum Einsatz kommen (keine dynamische Speicherverwaltung).

- **non-discriminated unions**: Es wird zum Beispiel aus einer union ein Zeiger ausgelesen, obwohl ein Integer hineingeschrieben wurde.
  Lösung: nur discriminated Varianten verwenden (oder Objekte)

Abstract Data Types

Beispiel in C++

```
class point {
public:
      point(int a, int b) { x = a; y = b; }
      void x_move(int a) { x += a; }
      void y_move(int b) { y += b; }
      void reset() { x = 0; y = 0: }
private:
      int x, y;
};
```

Es wird nicht direkt, sondern über Methoden auf die Werte zugegriffen. Auf diese Weise können Implementierungsdetails leicht geändert werden, ohne dass dies nach außen sichtbar wird.

Generic Abstract Data Type (C++)

Template in C++:

```
template<class T> class Stack {
public:
      Stack(int sz)           { top = s = new T[size = sz]; }
      ~Stack()                { delete[] s; }
      void push(T el)         { *top++ = el; }
      T pop()                 { return *--top; }
private:
      int size;
      T *top, *s;
}

void foo() {
      Stack<int> int_st(30);
      Stack<item> item_st(100);
      ...
}
```

T ist in diesem Fall ein **Typparameter**, also ein Name, welcher für einen Typ steht. Bei der Verwendung müssen für Typparameter konkrete Typen angegeben werden.
Generische Funktionen arbeiten im Unterschied dazu mit Typinferenz und können feststellen, welcher Typ für einen Typparameter stehen soll.

Strong and Static Type Systems

Ein Typsystem ist ein System von Regeln, welches Verbindungen zwischen den Typen und Ausdrücken beschreibt. Durch dieses Regelsystem wird spezifiziert, was man unter konsistenten Typen versteht.

Ein **starkes**[2] Typsystem sichert Typkonsistenz zu.
Dies kann allerdings nur erreicht werden in dem statische Typüberprüfungen durchgeführt werden. Überprüfung zur Laufzeit reichen nicht aus, da nicht garantiert werden kann, dass der Teil des Codes welcher einen Fehler enthält auch tatsächlich ausgeführt wird.

Es ist aber nicht immer notwendig ein starkes Typsystem zu haben. Argumentation: „Wen stört es wenn in einem Teil eines Programms, der niemals ausgeführt wird ein Typfehler ist?“

Ein **statisches** Typsystem kennt die Typen von Ausdrücken zur Compile- Zeit. Dies hat allerdings nichts damit zu tun ob auch Überprüfungen durchgeführt werden, auch wenn dies meist gemacht wird. Deswegen ist ein statisches Typsystem auch fast immer stark.

Type Compatibility

Es gibt mehrere Arten der Typkompatibilität (auch conformance, equivalence):

Namens Kompatibilität: Haben 2 Typen denselben Typnamen, so sind dies gleich, bzw. kompatibel. Es gibt folglich nur eine Stelle an der ein Typ deklariert wurde (es können nicht an verschiedenen Stellen Typen mit dem gleichen Namen deklariert werden, welche dann als „gleich“ angesehen werden).
Struktur Kompatibilität: Typen sind dann kompatibel, wenn sie dieselbe Struktur aufweisen. Es können hier Typen auch an unterschiedlichen Stellen im Programm eingeführt werden. Sobald sie dieselbe Struktur haben, sind sie kompatibel.

Namenskompabilität ist wesentlich stärker als Strukturkompabilität, da mit ihnen mehr ausgedrückt werden kann. Dadurch das Namen gleich sein müssen, bekommen diese eine Bedeutung, sie „stehen für etwas“. Wenn zum Beispiel der Name *sorted list* verwendet wird, impliziert dieser bereits, dass er für eine sortierte Liste steht. Der Compiler kann zwar nicht prüfen, dass die Elemente darin tatsächlich sortiert sind, aber es ist sicher gestellt, dass Instanzen von *sorted list* immer dieselben Eigenschaften aufweisen (es kann keine falsche *sorted list*, welche diese Eigenschaften nicht aufweist hinein geschummelt werden).

2 Es werden auch Begriffe wie “stark bis auf ...” verwendet

Fast jede Sprache verwendet sowohl Namens- als auch Strukturkompabilität nur in unterschiedlichem Umfang.

C ist dafür bekannt auf strukturelle Gleichheit zu setzen. Namenskompabilität spielt eine untergeordnete Rolle und ist nur bei *structs* [sic] relevant.

Ada verwendet fast ausschließlich Namenskompabilität, in den älteren Varianten (Ada 83) war dies sogar die einzige Art der Kompatibilität.

Beispiel: obwohl beide Typen strukturell gleich sind, sind sie in Ada nicht kompatibel. IA und IB sind beides Instanzen.

```
IA: array (1..100) of INTEGER;
IB: array (1..100) of INTEGER; -- not compatible with IA
```

Structural Type Compatibility

```
type s1 is struct {
int x; int y;
}

type s2 is struct {
int x; int y;
}

type s3 is struct {
int y; int x;
}

type s4 is struct {
int y;
}
```

Im Falle von struktureller Typäquivalenz wären s1 und s2 gleich. Was wäre aber wenn die beiden Variablen x und y vertauscht werden würden? Zählt der Name oder die Reihenfolge?
Es gibt mehrere Definitionen von struktureller Äquivalenz, manche die s2 und s3 als gleich sehen würden und andere die nicht.
Eine Form der Kompatibilität würde sogar zulassen dass s3 und s4 kompatibel wären, wenn in s3 nur auf das y Element zugegriffen werden würde (nur dieses benötigt werden würde). Dies entspricht dem Ersetzbarkeitsprinzip in der Objekt orientierten Programmierung (s4 wäre ein Obertyp von s3).
Dies lässt sich auch in Namenskompabilität umwandeln indem etwas wie Vererbungsstrukturen angewandt werden (s3 erbt von s4)

Type Coercion

Type Coercion ist die implizite Form von Typumwandlung.
Mit dieser Technik kann mit 2 unterschiedlichen Typen so umgegangen werden, als wären diese äquivalent.

Beispiel in C:

```
int x;
float z;
...
x = x + z;              // coercion of x to float, result to int
x = x + (int)z;         // explicit type cast, no coercion
```

x wird zuerst durch coercion in einen float wert umgewandelt. Es folgt die Fließkommazahl-Addition von x und z. Das Ergebnis wird dann erneut durch coercion in einen Integer umgewandelt, die Komma Stellen abgeschnitten.
Um ein anderes Verhalten zu erzwingen muss explizit gecastet werden (Zeile 2).

Es wurde auch ausführlich untersucht ob sich derartige implizite Umwandlungen als Ersatz für das Ersetzbarkeitsprinzip anbieten. Dies ist aber nicht möglich, da Typumwandlung wesentlich schwächer ist (kein dynamisches Binden). Dies beweist das Aufgaben die durch dynamisches Binden gelöst werden im Allgemeinen nicht statisch lösbar sind.

Names and Constraints in Ada

Zur Zeit von Ada war der Begriff „Subtyping“ wie er heute verwendet wird noch nicht geläufig.Ein **Subtype** in Ada schränkt einen bestehenden Typ zusätzlich ein.

```
type IntVector is array (Integer range <>) of Integer;
type VarRec (IsChar: Boolean) is
      record X: Float;
            case IsChar of
                  when False => Y: Integer;
                  True => U: Char;
            end case;
      end record;

subtype Vec100 is IntVector(0..99);
-- compatible with IntVector
subtype VarRecChar is VarRec(True);
-- compatible with VarRec
subtype MyInt is Integer range -9..9;
-- compatible with Integer
```

Der Typ *IntVector* wird als Integer Array mit offenen Grenzen (dargestellt durch „<>“) deklariert. Der eingeschränkte Subtype *Vec100* welcher nur Werte von 0-99 enthalten darf ist zum Typ *IntVector* kompatibel.

Der Varienten-Record *VarRec* mit *isChar* als Diskriminante (bestimmt welches Element im Record ist) ist ebenfalls kompatibel mit dem subtype *VarRecChar*, bei welchem *isChar* immer *true* ist.

Der Subtype MyInt schränkt die Basisklasse Integer auf Werte von -9 bis 9 ein und bleibt gleichfalls kompatibel zu Integer.

Später wurde erkannt, dass sich dieses Konzept nicht für Untertypbeziehungen in der OO Programmierung eignet. Das Arbeiten mit Subtypes ist **nicht statisch prüfbar**.

Polymorphic Types

Monomorph: Jeder Ausdruck, jede Variable hat genau einen Typ, welcher eindeutig bestimmt ist.
Polymorph: Variablen und Ausdrücke können gleichzeitig mehr als nur einen Typ haben. In Java kann ein Variable beispielsweise einen deklarierten Typ haben, welcher sich vom dynamischen unterscheidet.

Arten des Polymorphismus:

- universeller Polymorphismus
  - parametrischer Polymorphismus (Generezität): eher statisch
  - inclusion Polymorphismus (Subtyping): eher dynamisch

- ad-hoc Polymorphismus
  - Overloading
  - Coersion (Umwandlung)

Scalar Types in Ada

Definieren eines einfachen Typs:

```
type Small_Int is range -10..10;    -- user-defined integer
```

Small_Int ist hier ein eigenständiger Typ, kein Subtype. Die Maschinenrepräsentation bestimmt der Compiler, sie ist in diesem Fall nicht vorgegeben.

```
type Smaller_Int is range -8..7;
```

In diesem Fall könnte angegeben werden, dass der Typ *Smaller_Int* nur 4 Bit Speicher benötigen darf, was dementsprechend die Maschinenrepräsentation vorgibt.

```
type Two_Digit is mod 100;          -- modulo numb. (0..99)
```

Der Type Two_Digit beinhaltet werte von 0-99, wobei 99+1 = 0 (modulo). Wird beim (zuerst definierten) Typ *Small_Int* 10+1 gerechnet, wird eine Exception geworfen.

```
subtype Natural is Integer range 0..INTEGER'LAST;
subtype Positive is Integer range 1..INTEGER'LAST;
```

Subtypes sind keine eigenständigen Typen.

```
type Celsius is new Integer;        -- incompatible to Integer
type Farenheit is new Integer;
```

Diese Typen sind eigenständig, aber (bzw. genau deshalb) inkompatibel zu deren „Eltern-Typen“. Der Versuch eine Fahrenheit Einheit zu einer Celsius oder Integer zu addieren führt zu einem Typfehler. Es könnte aber gecastet werden: Fahrenheit → Integer → Celsius.

```
type MyFloat is digits 10; -- user-defined floating point
```
Selbstdefinierte Fließkommazahl *myFloat* verlangt das mit ihr 10stellig gerechnet werden kann.

```
type Fix_Pt is delta 0.01 range 0.0..100.0; -- fixed point
```
Selbstdefinierte Fixkommazahl Fix_Pt mit einer Genauigkeit von 0.01 (*delta 0.01*). Der Fehler bleibt im gesamten Wertebereich immer gleich (nicht wie bei Fließkommazahl)

```
type Dec_Pt is delta 0.01 digits 3; -- accurate fixed point
```
Hier wird zusätzlich zur Genauigkeit die Anzahl der Stellen angegeben (*digits 3*). Dies heißt aber auch, dass nicht genauer als angegeben gerechnet werden darf (was im Obigen Beispiel sehr wohl möglich ist). Solche Typen sind vor allem im Finanzbereich von Bedeutung.

```
type Week is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```
Aufzählungstyp Week

```
type Daily is array (Week) of Integer;
```
Daily ist Subtype von Week (ohne Einschränkungen)

```
type At_Work is array (Week range Mon..Fri) of Integer;
```
Daily ist Subtype von Week mit der Einschränkung, dass er nur die Tage Mon – Fri enthält.

Discriminated Union in Ada

Anhand der Diskriminante wird unterschieden welcher Wert im Record gespeichert werden darf.

```
type Address_Type is (Absolute, Offset);
type Safe_Address is
      record (Kind: Address_Type := Absolute)
            case Kind is
                  when Absolute =>
                        Abs_Addr: Natural;
                  when Offset =>
                        Off_Addr: Integer;
            end case;
end record;
```

In Ada wird zwischen *mutable* und *immutable* Discriminated Unions unterschieden. Immutable Discriminated Unions erlauben keine Änderungen der Diskriminante.

Pointers in Ada

Hier kann der größte Nachteil von Ada, die lange Syntax erkannt werden.

```
T: Tree_Ref; //Zeiger auf einen Baum, wurde bereits definiert
...
T := new Bin_Tree_Node; //neue Instanz (am Heap)
T.all := (Info => 0, Left => null, Right => null)
```

Mit *new Bin_Tree_Node* wird eine neue Instanz am Heap angelegt. Wird einfach eine Variable deklariert, wird der Speicher dafür am Stack reserviert.
T.all ist eine explizite Dereferenzierung: Es soll nicht in T, sondern in das auf was die Variable T zeigt geschrieben werden.

```
type Message_Routine is access procedure(M: String);
Give_Message: Message_Routine;
...
Give_Message := Print_This'Access;
Give_Message.all("This is not an error");
```

In aktuelleren Varianten von Ada können auch Zeiger auf Routinen vergeben werden (ähnlich C).
'Access → liefert die Adresse auf welch der Zeiger zeigt.
Give_Message.all("This is not an error") ruft die Routine mit dem Argument *"This is not an error"* auf.

```
Structure: array (1..10) of aliased Component; //Array
type ComponentPtr is access all Component;     //Zeiger auf Inhalt
                                               //des Arrays
Mine, Yours: ComponentPtr;                     //Instanzen von ComponentPtr
...
Mine := Structure(1)'Access;
Yours := Structure(2)'Access;
```

Adressen können nur von Typen bezogen werden die dies auch erlauben. Das Schlüsselwort *aliased* erlaubt eben dies.
In älteren Versionen von Ada war das Beziehen von Adressen generell nicht möglich.

Modularity and Programming in the Large

Packages In Ada

Package Specification in Ada

```
package Dictionary is
      procedure Insert(C:String; I:Integer);
      function Lookup(C:String) return Integer;
end Dictionary;
```

Bei einer Package Definition (Modul) in Ada wird angegeben welche Schnittstellen dieses Packages nach außen sichtbar sein sollen (entspricht in etwa einem Interface in Java).
Zu einer Package Definition existiert genau ein Package Body (teilen desselben Namen), also die Implementierung der Package Funktionalität.

Package Body in Ada

```
package body Dictionary is
      type Node;
      type Node_Ptr is access Node;
      type Node is record
                  Name: String;
                  Id: Integer;
                  Next: Node_Ptr;
            end record;
      Root: Node_Ptr;
      procedure Insert(C:String; I:Integer) is begin ...
      end Insert;
      function Lookup(C:String) return Integer is begin ...
      end Lookup;
begin
      Root := null;
end Dictionary;
```

Der letzte Teil in der obigen Implementierung (*begin – end*) dient dazu die variable *Root* zu deklarieren (eine Art Konstruktor).

Use of Package in Ada

```
with Dictionary;        //verwende das Packet “Dictionary”
use Dictionary;         //ermöglicht unqualifizierte Verwendung
procedure Main is
      Code: Integer;
begin
      Insert("volleyball", 1);
      Insert("football", 2);
      ...
      Code := Lookup("football");
      ...
end Main;
```

```
//Verwende X in Packagedef.
with X;
package T is
      C: Integer;
      procedure D(...);
end T;

//hier kann auch auf X zugegriffen werden
package body T is
      ...
      ...
      ...
end T;
```

```
with T;
procedure U(...) is
      ...
      //qualifizierter Zugriff
      ... T.D(...) ...
      ... T.C ...
end U;

with T;
use T;
procedure U(...) is
      ...
      //unqualifizierter Zugriff
      ... D(...) ...
      ... C …
end U;
```

Packages welche in der Packagedefnition eingebunden werden können auch im Body verwendet werden. Da Schlüsselwort *with* bindet Packages ein. Bei deren Verwendung muss jedoch immer qualifiziert zugegriffen werden. Möchte man dies nicht kann man mit dem Schlüsselwort *use* ein bereits eingebundenes Package für den unqualifizierten Zugriff freigeben. Verwenden 2 Packages gleiche Namen und werden beide mit *use* eingeführt, wird von use nur der Teil sichtbar gemacht, welcher eindeutig ist. Auf mehrdeutige Namen muss weiterhin qualifiziert zugegriffen werden.

Package Specification with Private Part

Die Grundidee von Packages war, dass zuerst ein Deklarationsteil geschrieben wird und dieser bereits im Code verwendet werden kann, bevor die Implementierung geschrieben wurde.
Der Compiler sollte mit den Informationen aus dem Deklarationsteil auskommen. Werden allerdings spezielle Datenstrukturen in der Implementierung verwendet muss der Compiler bereits wissen, wie viel Platz er für diese reservieren muss (er muss deren Typ kennen). Aus diesem Grund existiert auch im Deklarationsteil ein „privater“ Abschnitt, welcher Informationen enthält welche der Compiler kennen muss, auf diese aber trotzdem nicht von außerhalb zugegriffen werden kann[3]. Dieser enthält die nötigen Typdefinitionen.

```
package Dictionary is
      type Dict is private;
      procedure Insert(D: in out Dict; C: String; I: Integer);
      function Lookup(D: Dict; C: String) return Integer;
private
      type Node;
      type Node_Ptr is access Node;
      type Node is record
                  Name: String;
                  Id: Integer;
                  Next: Node_Ptr;
            end record;
      type Dict is new Node_Ptr;
end Dictionary;
```

In Java ist es gar nicht möglich private Typen zu definieren. Da nur Zeiger eingesetzt werden muss der Compiler auch keinen Speicher für ganze Typen reservieren.

3 In den Vorherigen Beispielen wurden in den Prozeduren nie “private” Datentypen (wie das Dictionary), sondern immer nur globale Typen wie String, Integer,... verwendet.

Module in ML

ML ist eine funktionale Sprache aus der Mitte der 80er Jahre. Ein *struct* entspricht hier einem Modul oder Package.
struct ist das Beginnstatement für *structure* (die Kategorie eines Namens)
Ein Name (Dictionary), welcher ein struct enthält wird eingeführt.

```
structure Dictionary =
struct                        //ab hier beginnt der Inhalt des structs
      exception NotFound;
      val create: (string * int) list = nil;

      fun insert(c:string, i:int, nil:(string*int)list) = [(c,i)]
      |     insert(c, i, (cc,ii)::cs) =
                  if c = cc then (c,i)::cs
                              else (cc,ii)::insert(c,i,cs);

      fun lookup(c:string, nil:(string*int)list) = raise NotFound
      |     lookup(c, (cc,ii)::cs) =
                  if c = cc then ii
                              else lookup(c,cs);
end;
```

Das Struct enthält:

- eine Exception
- eine Konstante create mit dem Typ (string * int) list = nil;
  Wobei (string * int) ein Tupel bestehend aus einem String und einem Int ist.
  (string * int) list steht dann für eine Liste dieser Typen.
- Eine Funktion insert: hier wird unterschieden ob die Liste leer ist (dann soll sie genau das übergebene Paar enthalten) oder nicht leer (dann wird das Paar an der richtigen Stelle eingefügt)
- Eine Funktion lookup

Use of Module in ML

```
val d = Dictionary.create
val e = Dictionary.insert("X", 7, d);
...
Dictionary.lookup("X", e);
```

Signature in ML

```
signature DictLookupSig = sig
      exception NotFound;
      val lookup: string * (string * int) list -> int
end;
```

Module Constrainted by Signature in ML

```
structure LookupDict: DictLookupSig = Dictionary;
val l = LookupDict.create;               (* not allowed *)
val d = Dictionary.create;
val e = Dictionary.insert("X", 7, d);
val v = LookupDict.lookup("X", e);
val k = LookupDict.insert("Y", 8, e);    (* not allowed *)
```

Generic Module in ML

```
structure Dictionary =
struct
      exception NotFound;

      val create = nil;

fun insert(c, i, nil) = [(c,i)]
|     insert(c, i, (cc,ii)::cs) =
            if c = cc then (c,i)::cs
                        else (cc,ii)::insert(c,i,cs);

fun lookup(c, nil) = raise NotFound
|     lookup(c, (cc,ii)::cs) =
            if c = cc then ii
                        else lookup(c,cs);
end;
```