Emerald

Entworfen an der Universität von Washington 1985 - 1987

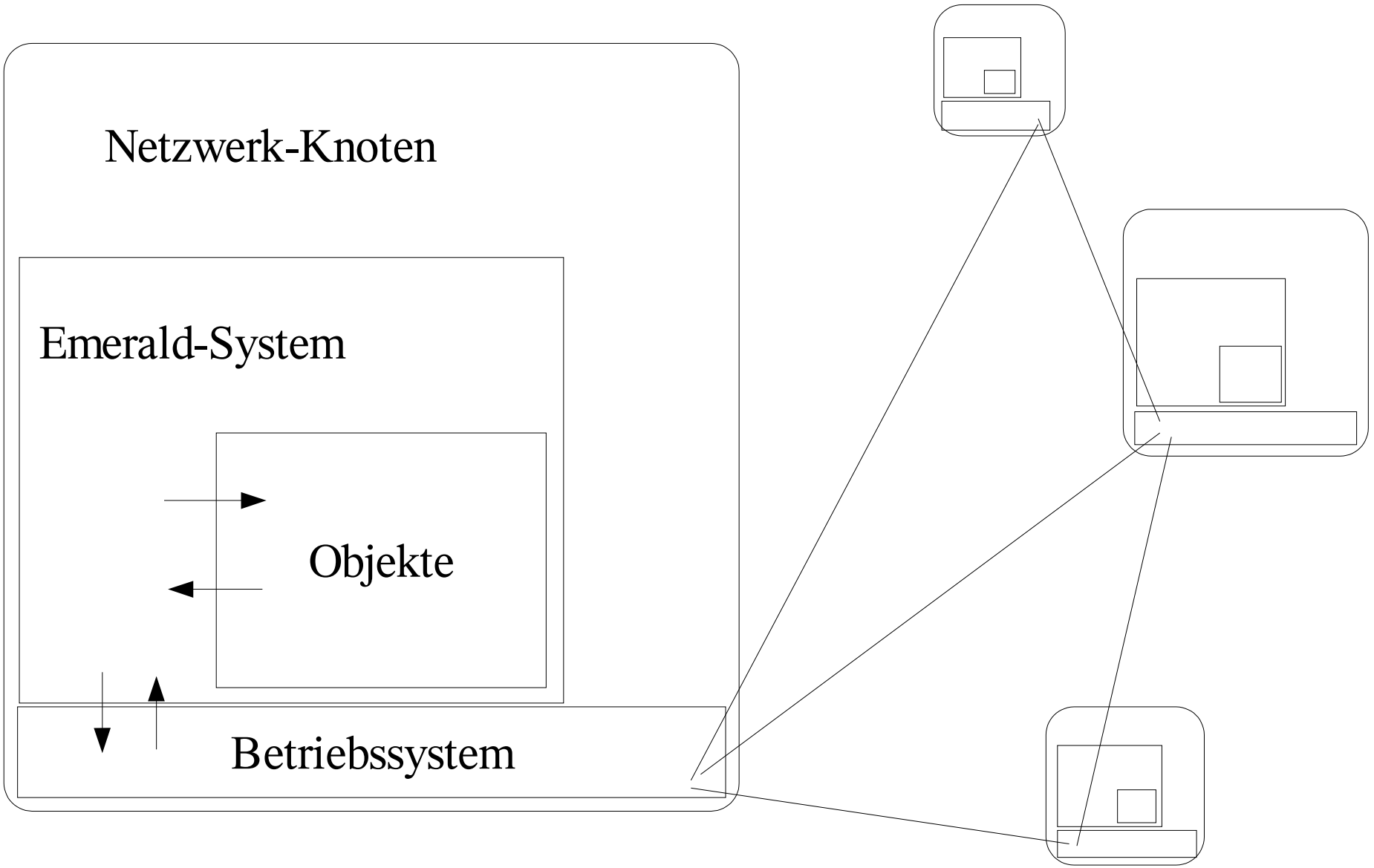
Design und Implementierung einer objektorientierten Programmiersprache mit besonderen Ansprüchen.

Vorwiegend als Machbarkeitsstudie

- Es scheint keine Applikationen (bis auf ein Mail-System) zu geben
- Nur kleine Codefragmente für Performance-Messungen

P.S. Etwa gleichzeitig zu C++ entstanden

Schema eines Netzknotens



Übersicht

- Entwurfsziele
- Objekte / Typen
- Mobilität
- Zusicherungen / Hinweise an das Emerald-System
- Implementierung (Nutzung von Zusicherungen)
- Besonderheiten der Sprache (Im Vergleich zu Java / C++)

Übersicht

- Entwurfsziele
- Objekte / Typen
- Mobilität
- Zusicherungen / Hinweise an das Emerald-System
- Implementierung (Nutzung von Zusicherungen)
- Besonderheiten der Sprache (Im Vergleich zu Java / C++)

Entwurfsziele:

- Unterstützung zur Entwicklung verteilter Anwendungen
  - Mobilität von Objekten
- Unterstützung von Nebenläufigkeit
  - Prozesse
- Objektorientierter Ansatz
- Netztransparenz
- Einheitliches Typkonzept (es gibt nur Objekte)
- **Hohe Performance**

Effizienz durch:

- Möglichst viele Information zur Compilezeit nutzen
- beim Design der Sprache auf die effiziente Implementierbarkeit achten
  - Sprachkonstrukte, mit denen der Programmierer dem Compiler / dem Emerald-System optimierungsrelevante Hinweise geben kann

Übersicht

- Entwurfsziele
- Objekte / Typen
- Mobilität
- Zusicherungen / Hinweise an das Emerald-System
- Implementierung (Nutzung von Zusicherungen)
- Besonderheiten der Sprache (Im Vergleich zu Java / C++)

Objekte

- Nur über exportierte *operations* / *functions* ansprechbar
  - *operation*: darf Seiteneffekte haben
  - *function*: darf keine Seiteneffekte haben
- Keine Klassen und keine Vererbung
  - ➔ Konstruktor-Objekten zum Erzeugen von Objekten
- Netzwerkweit eindeutig identifizierbar
- Keine Kopien (immer Referenzen)
  - ➔ Keine Konsistenzprobleme
  - ➔ *Call-By-Refernce*

```
const IntegerNodeCreator ← immutable object INC
  export new                                    % die zu exportierende Operation

  const IntegerNodeType ← type INType           % Typ-definition
    function getValue ← [Integer]
    operation setValue [Integer]
  end INType

  operation new[val : Integer] → [aNode : IntegerNodeType]
    aNode ← object IntegerLiteral               % zuweisung des ergebisses
      export getValue, setValue                 % definition der Objektschnittstelle
      monitor                                   % Synchronisation für Nebenläufigkeit
        var value ← val
        operation getValue → [v: Integer]       %Implemantation des Schnittstelle
          v ← value
        end getValue
        operation setValue [v : Integer]        %Implemantation des Schnittstelle
          value ← v
        end setValue
      end monitor
    end IntegerLiteral
  end new
end INC
```

Typ eines Objekts:

Ein Objekt hat prinzipiell den Typ seiner Erzeugung, „implementiert“ aber alle Typen, deren Schnittstelle er erfüllt.

- Keine explizite Zuweisung wie in Java durch *extends* / *implements*
  - ➔ Keine direkte Kontrolle über die Verwendbarkeit von Objekten durch den Programmierer
- Problem: Typen haben keine semantische Bedeutung (sind rein syntaktisch)

Typ

Ein Typ besteht aus dem Typnamen, einer Liste von Operationsnamen und deren Parametersignaturen.

- Typen der Übergabeparameter **und** er Rückgabewert**e** sind signifikant

- Auch Typen sind Objekte und können als Parameter übergeben werden
  ➔ Nutzung, wie bei den C++ Templates (/Java 1.5) möglich

```
immutable type AbstrctType
    function getSignature → [Signature]
end AbstrctType
```

Stack - Implementierung

```
const Stack == immutable object aStackCreator
    export of
    function of[eType : AbstractType] → [result : stackType]
        where
            stackType == type stackType
                operation Push[eType]
                operation Pop → [eType]
                function Top → [eType]
                function Empty → [Boolean]
            end stackType
        end where
        result ← object aStack
            % representation declarations
            operation Push[anElement : eType]
                ⋮
            end Push
            ⋮
        end aStack
    end of
end aStackCreator
```

Figure 3.1: A polymorphic stack

Sorted Collection

```
const Sortable == immutable type Sortable
    function <[Sortable] → [Boolean]
end Sortable
const SortedCollection == immutable object aSortedCollectionCreator
    export of
    function of[eType : AbstractType] → [result : collectionType]
        where
            collectionType == type collectionType
                operation Add[eType]
                operation getElement[Integer] → [eType]
                function Size → [Integer]
            end collectionType
            eType ∘> Sortable
        end where
        result ← object aCollection
            % representation declarations
            operation Add[anElement : eType]
                ⋮
            end Add
            ⋮
        end aCollection
    end of
end aStackCreator
```

Figure 3.2: A polymorphic sorted collection

Typkonformität

Wann erfüllt ein Objekt einen Typ?

1. Das Objekt exportiert namentlich alle Operationen des Typs

2. Die Parameter der Operationen sind gleich oder allgemeiner als die der Typdefinition

3. Die Typen der Rückgabeparametern sind gleich oder spezieller als die der Typdefiniton

Die Relation Objekt X ist typkonform zu Objekt Y ist *reflexiv* und *transitiv*. $X \triangleright Y$

Polymorphismus

- Verschiedene Implementierungen von Objekten eines Typs

- Gleichnamige Operationen, die sich nur durch ihre Parametersignatur unterscheiden
  (auch Typ der Rückgabewerte relevant)
  (Im Gegensatz zu Java und C++)

  Wenn bei einem Aufruf mehrere Operationen in frage kommen, wird die mit der Speziellsten Signatur verwendet.

Übersicht

- Entwurfsziele
- Objekte / Typen
- Mobilität
- Zusicherungen / Hinweise an das Emerald-System
- Implementierung (Nutzung von Zusicherungen)
- Besonderheiten der Sprache (Im Vergleich zu Java / C++)

Mobilität

Eine wichtige Eigenschaft für Verteilte Systeme ist die Mobilität.

Ziele:

- Lastausgleich
- Nutzung spezieller Hardware
- Nur so völlige Netztransparenz möglich

Arten der Mobilität

Aus der Sicht eines Betriebssystems gibt es zwei Arten von Mobilität:

grobkörnig: ganze System-Prozesse wandern

feinkörnig: Teile eines Prozesses wandern
(bei Emerald die Objekte)

(feinkörnige Mobilität wurde erstmals in Emerald implementiert)

Probleme der Mobilität

- I/O bleibt stationär (Festplatte, Ausgabegeräte, ...)

Wenn sich Objekte bewegen:

- Lokale Operationsaufrufe werden plötzlich zu entfernten u.U.
- Alle Referenzen auf das Objekt müssen angepasst werden

Häufige Antwort auf dieses Problem ist es alle Aufrufe als entfernte Aufrufe zu behandeln.

Da der Fall lokaler Aufrufe der weitaus häufigste ist, kann der Effizienzverlust, der entstehen würde nicht hingenommen werden.

- Das Problem der sich verändernden Aufrufe beim Bewegen von Objekten werden wir später näher betrachten.

Auch das Bewegen von Prozessen wird uns später noch beschäftigen.

Probleme der Mobilität II

Wenn sich Prozesse bewegen:

- Der Prozeß muß angehalten
- Auf den Zielknoten verschickt
- Dort an der gleichen Stelle fortgesetzt werden.

P.S. Wer werden sehen, daß dies bei nichtnaiver Implementierung zu erheblichen Problemen führt.

Prozesse in Emerald

Ein Objekt kann optional einen Prozeß enthalten, der bei der Erzeugung des Objekts asynchron gestartet wird.

Solche Objekte werden *aktiv* genannt.
Objekte ohne Prozeß *passiv*.

Auf diese Art entsteht die Nebenläufigkeit in Emerald.

Synchronisation wird durch *Monitore* realisiert

Übersicht

- Entwurfsziele
- Objekte / Typen
- Mobilität
- Zusicherungen / Hinweise an das Emerald-System
- Implementierung (Nutzung von Zusicherungen)
- Besonderheiten der Sprache (Im Vergleich zu Java / C++)

Mobilität von Objekten II

(move, fix, unfix, refix)

Einflußnahme auf der Position von Objekten durch den Programmierer:

- *Move* X *to* Y
- *Fix* X *at* Y
- *Unfix* X
- *Refix* X *at* Y

Vorsicht: die move-Anweisung darf vom Emerald-System ignoriert werden.

Sicher das gewünschte Verhalten liefert: (mit refix entsprechend)
Fix X *at* Y
Fix Y *at* X

- *locate* X liefert die aktuelle Position es Objekts X

Gruppenbildung von Objekten

Da es unsinnig ist, Objekte, die exklusiv innerhalb eines anderen Objektes existieren, von diesem getrennt zu bewegen, ist es möglich ein Objekt einem anderen Objekt als *attached* (zugehörig) zu deklarieren.

Das hilft dem Emerald-System bei der Entscheidung welche Objekte besser gemeinsam bewegt werden sollten

- ➔ Bessere Performance

Objekte II

Der Programmierer kann dem Compiler zusichern, daß ein Objekt *immutable* ist.
Damit weis der Compiler, daß das Objekt keine internen Daten beinhaltet, die sich über die Zeit ändern.

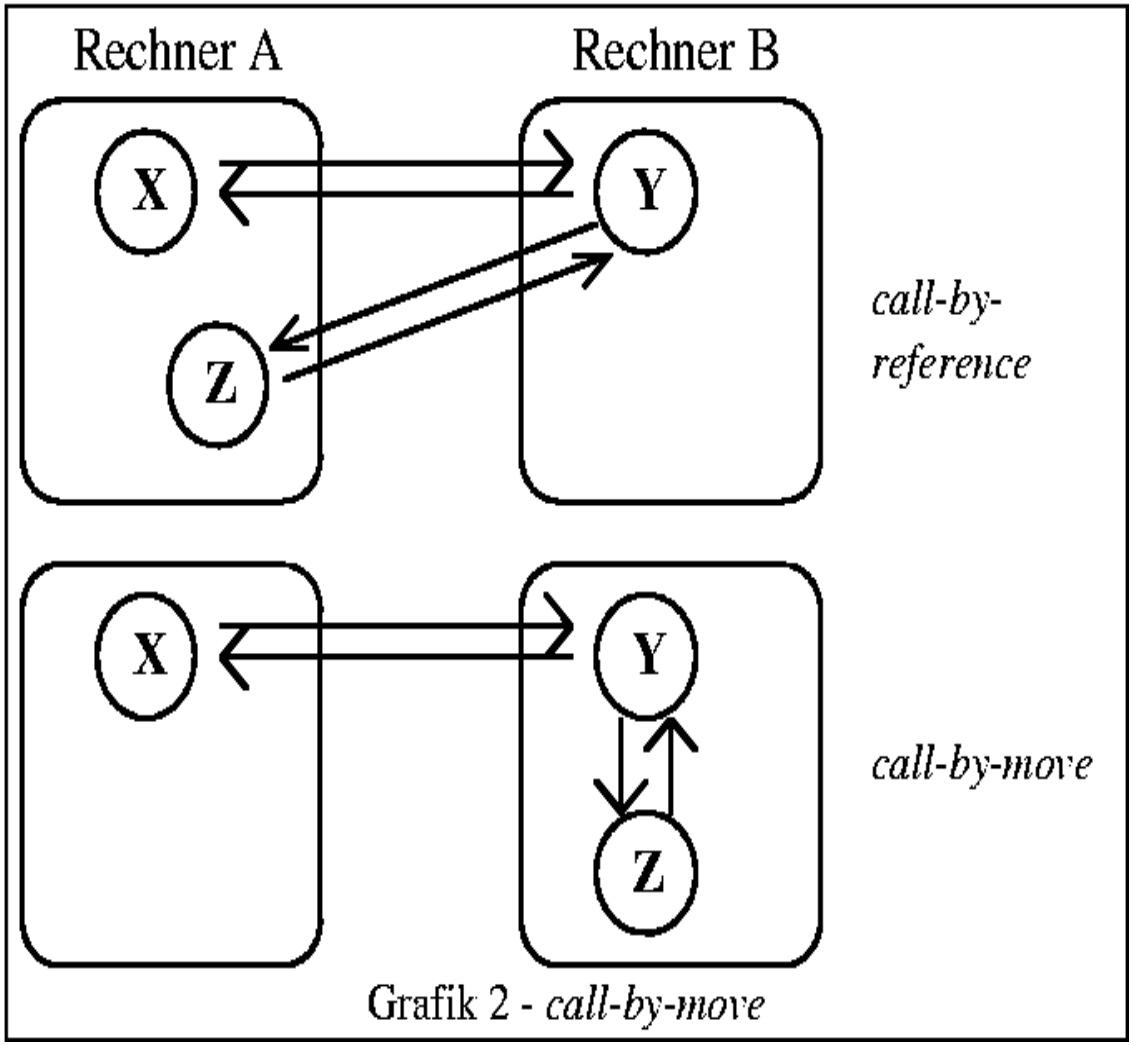
- Es dürfen vom Compiler / dem Emerald-System zu Optimierungszwecken Kopien des Objekts erzeugen werden ohne das Inkonsitenzen auftreten können

Varianten des Call-By-Reference

Als Entscheidungshilfen für das Emerald-System:

- Call-by-move:
- Call-by-visit:
- Call-by-move-return:

Können für jeden Parameter getrennt angegeben werden und sind für das Emerald-System unverbindlich.



Grafik 2 - *call-by-move*

Probleme verteilter Systeme I

Teile des Systems können ausfallen

➔ Objekte sind zeitweise oder nie mehr erreichbar

Das Emerald-System stellt hierzu *EventHandler* zur Verfügung, bei dehnen der Programmierer Informationen über

- ausfallende Knoten
- neu hinzugekommene Knoten
- Unerreichbare Objekte

- Fehlerhaft ausgeführte Operationen
  (z.B. divison by zero)

erhalten kann.

Übersicht

- Entwurfsziele
- Objekte / Typen
- Mobilität
- Zusicherungen / Hinweise an das Emerald-System
- Implementierung (Nutzung von Zusicherungen)
- Besonderheiten der Sprache (Im Vergleich zu Java / C++)

Implementierung I

Zur Entwicklung von Emerald gehörte auch eine Implementierung um die Design-Ziele zu überprüfen

(Damals unter einem UNIX-System auf einer **Micro CAX II**)

- Das Emerald-System besteht aus einem einzigen System-Prozeß (pro Netzknoten)
- Einem gemeinsamen Adreßraum für alle Objekte eines Knotens
- Der Compiler erzeugt direkt Maschinencode für alle Objekte

Implementierung II

Da die meisten Interaktionen lokal auf einem Netzknoten geschehen, werden diese in Emerald durch direkten Sprung/Zugriff auf Speicheradressen realisierte.

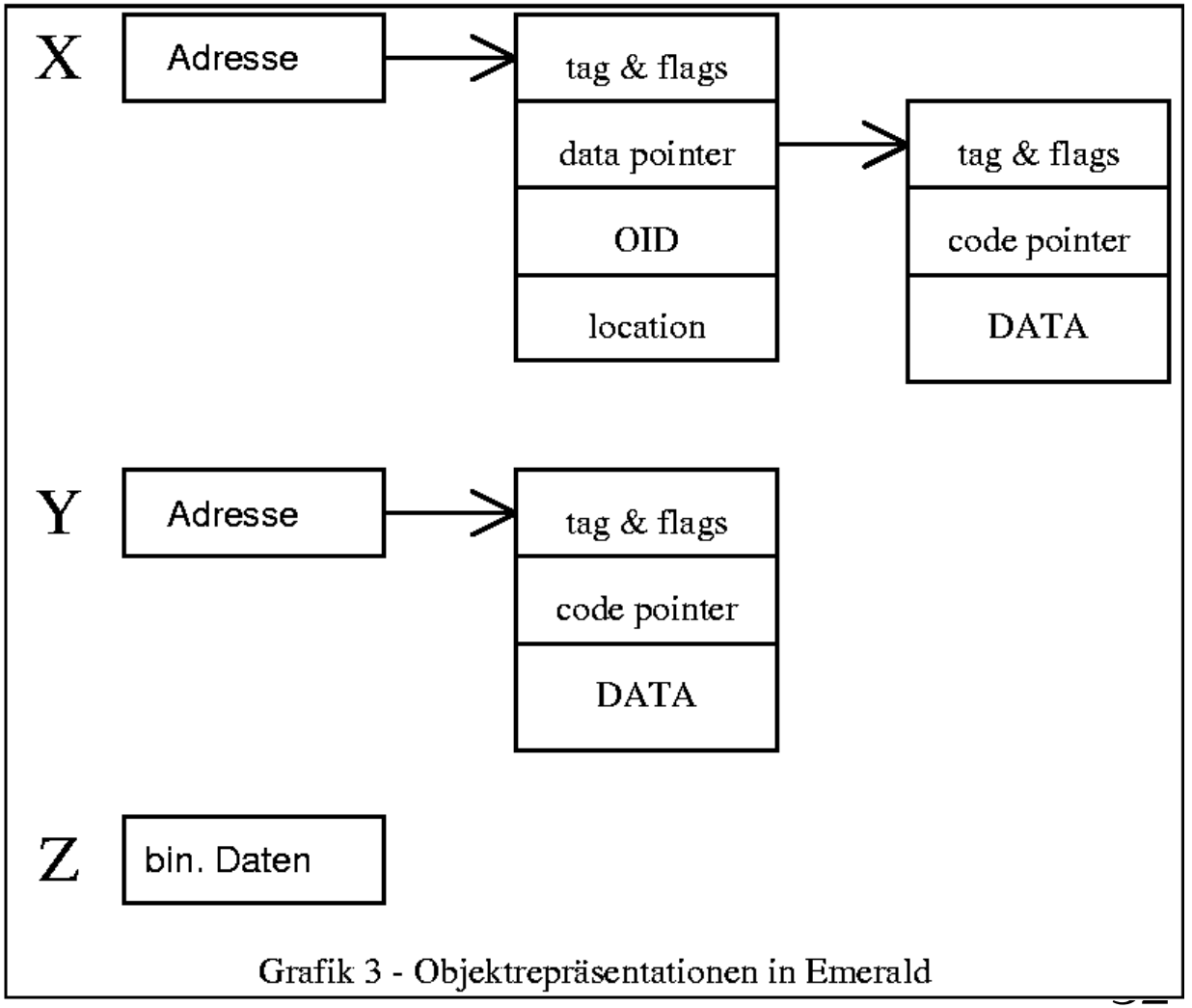
- ➔ Operationsaufrufe sind lokal genauso schnell, wie bei nichtverteilten prozeduralen Sprachen.

Implementierung III

Es werden 4 Klassen von Objekten unterschieden:

- Globale Objekte (X)
- Lokale Objekte (Y)
- Direkte Objekte (Z)

- *Immutable* Objekte



Grafik 3 - Objektrepräsentationen in Emerald

Implementierung IV(Mobilität)

Bewegt sich ein Objekt, so müssen:

1. alle gerade auf ihm ausgeführten Operation und der eventuell existierende innerer Prozeß angehalten werden.

2. alle auf dem Stack liegenden Operandenadressen ehemals lokaler Objekte, Referenzen auf das zu bewegende Objekt und Rücksprungadressen erkannt und Angepasst werden. (nicht trivial)

3. Schließlich kann das Objekt bewegt werden und auf dem Zielknoten das ganze analog (in Gegenrichtung) durchgeführt werden.

4. Angaben über die Position des Objekts auf beiden Rechnern aktualisieren

Implementierung V (Codesharing)

Ansatz in Emerald ist:

- Maschinencode ist Teil eines jeden Objekts

nun gibt es aber doch viele Objekte mit gleichem Maschinencode.

- ➔ Gemeinsam Nutzen zur Speicherersparnis
  - ➔ Code ist *immutable* Objekt des Objekts

- Beim Bewegen von Objekten wird der Maschinencode zunächst nicht mitverschickt
  - ➔ Weniger Netzwerk-Traffic
  - ➔ Geringere Übertragungszeiten

Impl. VI (Auffinden von Objekten)

Jeder Netzknoten verwaltet eine Liste aller im bekannten Objekte, deren Position im Netz und Verwaltungsinformationen.

Aufgrund der Nebenläufigkeit und der Latenz des Netzes können nicht alle Tabellen übereinstimmen.

Wenn eine Operation eines nichtlokalen Objekts aufgerufen wird:

- Wende dich an den Knoten aus deiner Tabelle
- Ist das Objekt nicht mehr dort, leitet dieser weiter
- Beim Objekt angekommen wird
    1. Das Ergebnis direkt zum Aufrufer gesendet
    2. Die Weiterleitungskette rückwärts durchlaufen und die Position des Objekts in den Tabellen aktualisieren

Implementierung VII

Die Emerald-Systeme verwenden zum Remote-Procedure-Call ein auf UDP aufgesetztes sicheres Protokoll.

Verschicken eines 'kleinen' Pakets ist fast genauso langsam, wie bei einem 'großen'.

- ➔ Möglichst ausnutzen der ca. 1500Bytes die möglich sind
- Passive Parameter-Objekte bis ca. 1000Bytes per *move* mitschicken. (Huckepack)
  - Verwaltungsaufwand des Bewegens rentiert sich fast immer durch spätere, dann lokale, Aufrufe

Aufgabe des Emerald-Compilers

Traditionell übersetzt ein Compiler 'nur' Programmtext in Maschinensprache.

Der Emerald-Compiler hat zusätzliche Aufgaben:

- Analyse des Programmtextes zum Erkennen der 4 Objekt-Klassen.
  ➔ Erzeugen optimierten Codes
- Entscheiden, ob noch Platz für *call by move* Huckepack
- Entscheiden, wann die Empfehlungen des Programmierers ignorieren sind

Performance relevante Design-Elemente

Hilfestellungen durch den Programmierer:

- Call-by-move / Call-by-visit / Call-by-move-return
- *move* X to Y (*fix / unfix / refix*)
- *function* als Sonderfall von *operation*
- *attached*
- *immutable*

Designentscheidung:

- Statische Typbindung (zur Compilezeit auflösbar)

Implementierungsentscheidungen:

- 4 Klassen von Objekten
- Ein System-Prozeß (Emerald-System)
- Gemeinsamer Adressraum aller Objekte eines Knotens
- Optimierung für den Fall lokale Objekte
- Huckepack call-by-move Parametertransport
- . . .

Übersicht

- Entwurfsziele
- Objekte / Typen
- Mobilität
- Zusicherungen / Hinweise an das Emerald-System
- Implementierung (Nutzung von Zusicherungen)
- Besonderheiten der Sprache (Im Vergleich zu Java / C++)

Emerald-Features I

- *widening / narrowing*
- Definieren von eigenen Operatoren aus

  ! # & * + - / < = > ? @ ^ | ~

- „Syntactic-Sugar“ :

  a=c&f entspricht *a=c.getF*

  c&f=a entspricht *c.setF[a]*

- *Forall:*

```
function identity[a: t] → [b : t]
   forall t
   b ← a
end indentity
```

Emerald-Features II

- Operator zum Typkonformitäts-Test
- Art 'Makros' zur Schreibersparnis
  - *class* erzeugt einen Objekt-Konstruktor-Obj. mit den einfach vererbten Operationen
  - *field*:

```
const field f : t ← init
```

wird zu:

```
const f : t ← init
export function getF → [ x : t]
  x ← f
end getF
```

```
field f : t ← init
```

wird zu:

```
f : t ← init
export function getF → [ x : t]
  x ← f
end getF
export function setF [ x : t]
  f ← x
end setF
```

Emerald-Features III

- *record:*

```
record aRecord
    a: Integer
    c: String
end aRecord
```

wird zu:

```
class aRecord [ xa : Integer, xc : String]
    field a : Integer ← xa
    field c : String ← xc
end aRecord
```

So erhält man ein Record mit get- und set-Methoden und passendem Objekt-Konstruktor

Was ich nicht erwähnt habe

- Garbage-Collection
- Mechanismen zum Umgang mit zeitweise und dauerhaft ausgefallenen Knoten
- . . .

Literatur / Quellen

Gute Quelle:

http://www.cs.ubc.ca/nest/dsg/emerald.html

- http://www.cs.ubc.ca/nest/dsg/emerald/papers/Report.ps (Hauptquelle)
- http://www.cs.ubc.ca/nest/dsg/emerald/papers/Primer.ps
- http://www.cs.ubc.ca/nest/dsg/emerald/papers/HutchinsonThesis.ps
- http://www.cs.ubc.ca/nest/dsg/emerald/papers/emeraldtypes.ps

Gute deutsche Quelle:

http://www.fpx.de/fp/Uni/Emerald/

als Postsctript: http://www.fpx.de/fp/Uni/Emerald/ps/Emerald.ps (2. Hauptquelle)

1-Seiten Kurzbeschreibung:

http://cui.unige.ch/SEMINAIRES/sem_31.html

Informatik-Bibliothek:

- Emerald: An Object-Based Language for distributed Programming (R 24349)
- TypecheckingPolymorphism in Emerald (R24376)

Implementierungen: HPUX-9/HPPA / FreeBSD-2.2/i386 / Linux-2.0.27/i386 / Solaris-2.5.1/Sparc / Windows95 / WindowsNT/i386

http://www.cs.ubc.ca/nest/dsg/emerald/dist/dist.html

http://www.diku.dk/forskning/distlab/Research/Internal/DistOOS/Emerald/Distribution.html

Anleitung zum Installieren und Compilieren: (von mir nicht genutzt)

http://www.rolemaker.dk/other/Emerald/

```
const Set ← immutable object Set
    export function of[eType : type] → [result : NewSetType]
        suchthat
            eType ▷
                immutable typeobject eType
                    function =[eType] → [Boolean]
                end eType
        where
            NewSetType ←
                immutable typeobject NewSetType
                    operation empty → [NewSet]
                    operation singleton[eType] → [NewSet]
                    operation create[sequenceOfeType] → [NewSet]
                end NewSetType
        where
            sequenceOfeType ←
                immutable typeobject sequenceOfeType
                    function lowerbound → [Integer]
                    function upperbound → [Integer]
                    function getElement[Integer] → [eType]
                end sequenceOfeType
        where
            NewSet ←
                immutable typeobject NewSet
                    function contains[eType] → [Boolean]
                    function +[NewSet] → [NewSet]
                    function *[NewSet] → [NewSet]
                    function -[NewSet] → [NewSet]
                    function cardinality → [Integer]
                end NewSet
        result ←
            object SetCreator
                export operation create[v : sequenceOfeType]→ [result : NewSet]
                    result ←
                        object NewSet
                            const repType ← Vector.of[eType]
                            var rep : repType

                            % The implementation of the operations and functions.
                        end NewSet
                end create
                export operation empty → [r : NewSet]
                    r ← self.create[nil]
                end new
                export operation singleton[e : eType] → [r : NewSet]
                    r ← self.create[e]
                end singleton
            end SetCreator
    end of
end Set
```