DERRIDA'S MACHINES PART III

ConTeXtures

Programming Dynamic Complexity

An Application:

FIBONACCI in ConTeXtures

© by Rudolf Kaehr

ThinkArt Lab Glasgow June 2005

"Interactivity is all there is to write about:
it is the paradox and
the horizon of realization."

FIBONACCI in *ConTeXtures*

Eine Abwendung

FIBONACCI in ConTeXtures

1 Genereller Rahmen der ConTeXtures

1.1 Wie sind die ConTeXturen einzuordnen?

1. Frage: Einzelsysteme

Was würden wir in Bezug auf Berechenbarkeit gewinnen, wenn wir 2 oder mehr völlig autonome und vollständige Pragramme auf ebenso autonomen und vollständigen Maschinen laufen lassen würden?

Antwort: Nichts, ausser dass mehrfach dasselbe realisiert würde.

2. Frage: Kommunikation

Was würde passieren, wenn diese autonomen und separierten Programme und Maschinen miteinander kommunizieren würden indem sie ihre Daten austauschen könnten?

Antwort: Der Gewinn wäre eine deutlich erhöhte Effizienz bezüglich Rechengeschwindigkeit, Arbeitsverteilung, Medienkapazitäten, usw.

Was wir nicht gewinnen würden, wäre eine Steigerung der *möglichen* Berechenbarkeit. Alles was die vernetzten Kommunikationssysteme leisten können, könnte, zumindest im Prinzip, durch einen einzigen Grossrechner geleistet werden.

3. Frage: Interaktion und Reflexion

Was würden wir gewinnen, wenn diese autonomen Rechensysteme nicht nur miteinander kommunizieren, sondern auch miteinander interagieren und auf ihre Interaktionen reflektieren könnten?

Antwort: Wir würden eine Komplexion erhalten, die sich nicht mehr auf ein einziges und einzelnes System reduzieren liesse, weil ein einzelnes und einheitliches System nicht mit sich selbst interagieren kann. Interaktion, und auch Reflexion, setzen Verschiedenheit der Systeme voraus. Dies würde echte Kooperation zwischen Systemen ermöglichen, die eine höhere Komplexität aufweist als ein noch so grosses Einzelsystem. Ko-kreation von gemeinsamen Umgebungen wären möglich. Für kommunikative Systeme gilt dies nicht, da sich deren Komplexität (im Prinzip) auf einen Informationsaustausch zwischen Teilsytemen eines einzelnen Gesamtsystems ohne Umgebung reduzieren lässt.

4. Vorschlag: ConTeXtures

ConTeXtures stellt das erste Programmierungsparadigma dar, das den Anforderungen einer interaktionalen und reflektionalen Komplexion von Rechnern und Programmen gerecht wird. Interaktivität und Reflektionalität von Computersystemen sind durch ConTeXtures konzeptionell erfassbar und werden durch diese programmiert.

Wie ist das möglich?

Weil ConTeXtures auf der Grundlage der polykontexturalen Logik konzipiert sind. ConTeXtures als Programmiersystem, besteht aus einer Distribution und Vermittlung der proto-typischen Programmiersprache ARS (Loczewski), die wiederum auf dem Lambda Kalkül aufbaut. Der Lambda Kalkül ist anerkanntermassen das Grundmodell jeglicher Programmierung überhaupt und ist äquivalent dem mehr Maschinen orientierten Modell der Turing Maschine.

ARS (Abstraktion, Referenz, Synthese) wurde von Lozcewski zwar als *proto-typisches System* eingeführt, ist aber von ihm auch für verschiedene Programmiersprachen, wie C, C^{++}, Scheme, Python zur Anschlussfähigkeit für Real-World-Programming weiter entwickelt worden.

ConTeXtures verstehen sich in der Tendenz dessen, was heute *"Interactional revolution in computer science"* (Milner, Wegner) genannt wird.

1.2 Was ist neu zu DERRIDA'S MACHINES?

ConTeXturen sind das erste Paradigma und System der Programmierung auf dessen Basis die konzeptionellen Entwürfe von DERRIDA'S MACHINES realisiert werden können. Es bietet zudem den programmtechnischen Anschluss zu bestehenden mono-kontexturalen Programmiersprachen, vermittelt durch die Anschlüsse von ARS, und damit eine Methode zu deren Distribution und Vermittlung.

ConTeXturen erlauben es, den *disseminantiven Schnitt* programmtechnisch zu realisieren. Der disseminative Schnitt ist die massgebliche Strategie, bestehende Programmiermethoden, -Konzepte und -Lösungen einer *Heterarchisierung* und damit einer Implementierung durch ConTeXtures zugänglich zu machen.

ConTeXtures zeigen jedoch ihre Stärke in einem Feld, das vorwiegend durch ambigue, polyseme und konfliktuöse Konstellationen gekennzeichnet ist und das von klassischen Paradigmen der Programmierung explizit ausgeschlossen ist. Programme müssen per definitionem disambiguiert werden. Lebensweltliche Verhältnisse, Sprache und Bedeutung, Interaktion und Reflexion, usw. sind als solche schlechtweg nicht disambiguierbar. Marvin Minskys Vorstoss zu einer Problemlösungsstrategie der *"multiple ways of thinking"*, p-Analogy, als Prototyp einer neuen Denkweise im Entwurf von intelligenten Systemen, wird durch den Einsatz der ConTeXtures einer Realisierung näher gebracht.

Konzeptionell wurde dieser Sachverhalt in aller Ausführlichkeit in *DERRIDA'S MACHINES* ausgeführt, *ConTeXtures* entwerfen zum ersten Mal ein dazu passendes Paradigma der Programmierung.

www.thinkartlab.com/pkl/media/DERRIDA'S MACHINES.pdf

2 Eine Metapher: Intelligentes Druckersystem

Beispiel eines autonomen reflektionalen und interaktionalen Echtzeit-Systems: ein MIRC-Druckersystem. (MIRC: mobile, interactional, reflectional, computation). Siehe auch: *Autonomic computing,* IBM.

Ein Drucker in einem komplexen Verbund von vernetzten Computern hat eine Fülle von Jobs zu erledigen. Heute wird dies durch eine statische Administration der Job-Hierarchie geregelt. Die Jobs sind in einer Warteschleife, die von aussen, durch einen Administrator, geregelt wird. Diese Prioritätenliste kann etwa nach dem zeitlichen eintreffen der Jobs, der grösse der Jobs oder der Priorität des Senders des Jobs, usw. geregelt werden.

Ein MIRC-Drucker ist ein lernfähiges System, das die Verhaltenspattern des Verbundes reflektiert, also seine Geschichte kennt und das durch Interaktion mit den Sendern in der Lage ist, Verhandlungen (Negotiations) durchzuführen, mit dem Ziel einer Verbund gemässen Optimierung der Prozeduren und Jobs zu gewährleisten. Und zwar während des Ablaufs der Jobs und nicht davor oder danach durch Anpassung der Prioritätenliste. Der menschliche Administrator eines solchen Systems hat nun die neue und weit reflektiertere Aufgabe, das Lernverhalten des Systems während des Veraufs zu pflegen und nicht die untere Ebene der Verteilung der Jobs, die nun vom System selbst geleistet wird.

Ein solches Druckersystem ist nun ein voll integriertes jedoch autonomes System des Gesamtverbundes und nicht bloss eine Peripherie.

Wie ist es möglich, ein solches intelligentes Druckerystem zu realisieren?

Reflektionalität und Interaktivität. Aufgrund der *Separierbarkeit* der Unentscheidbarkeit selbst-reflektionaler Systeme in distinkte distribuierte Systeme, wie in ConTeXtures realisiert, ist Reflektionalität und Interaktivität simultan, in Echtzeit, zum Prozessablauf möglich.

Was sind die Anforderungen an ein Interaktionssystem?

Algorithmische Systeme sind strukturell geschlossen und haben nur sekundär Zugang zu ihrer Umgebung (Turing Machine). Ihr Hauptziel ist die Optimierung der Berechnung der Algorithmen unter der Voraussetzung ihrer gesicherten Berechenbarkeit. *Interaktionssysteme* dagegen müssen bezüglich ihrer Reaktionsgeschwindigkeit optimiert werden. Je direkter ein System auf seine Umgebung reagieren kann, desto effizienter ist es. Reaktivität ist jedoch nicht identisch mit Rechengeschwindigkeit. *Real-time computing* ist primär abhängig von der Direktheit der Reaktivität und nicht von der Geschwindigkeit in der ein Algorithmus (zu Ende) berechnet wird, dessen Ausgangs-Situation (Daten, Annahmen) in der – noch so kurzen Zwischenzeit – längst obsolet geworden ist.

http://www.redherring.com/Article.aspx?a=12018&hed=Anticipating+Autonomics

3 Grundzüge der ConTeXtures

Als ein System der Dissemination (Distribution und Vermittlung) des Lambda Kalkül-basierten Paradigma der Programmierung ARS (Abstraction, Referenz, Synthese) mit seinem Konzept der Berechenbarkeit, sind die ConTeXtures durch die zwei grundsätzlich neuen Eigenschaften charakterisiert: *Reflektionalität* und *Interaktionalität*. Diese können als zwei Dimensionen der Dissemination von Algorithmen betrachtet werden und begründen die polykontexturale Matrix. Damit gehören die ConTeXtures zum dritten Typus der Programmierung.

Gemäss der Vermitteltheit der Teilsysteme der ConTeXtures ist die Aufbaustruktur durch Ebenen und Features charakterisiert, die in klassischen Systemen nicht existieren. Daraus ergibt sich folgende Architektonik.

3.1 Allgemeine Architektonik

Architektonik mit *Templates* und *Patterns*. Diese geben die Struktur des Systems bzgl. Reflektionalität und Interaktivität an. *Configurations* und *Constellations* regeln die verschieden Kombinationen der *Topics* (Datenstrukturen) und *Styles* (Programmierungsstile) von ARS Programmiersystemen. Eine explizite Thematisierung dieser Features kann hier nicht vorgenommen werden. Es sei auf den Text *ConTeXtures* verwiesen.

Dem Aufbau der Architektonik entsprechend erscheinen Fragestellungen, die die Datenstrukturen betreffen erst an später Stelle als *Configurations* und *Constellations*. Davor stehen die Strukturen der Vermittlung (Architektonik) und der Interaktionalität und Reflektionalität als *Templates* und *Patterns*. (siehe: Diagramme)

Beispiele, die die Leistungsfähigkeit von ConTeXtures beleuchten, können daher auf verschiedenen architektonischen Ebenen der ConTeXtures vorgeführt werden. Dabei sind 2 Grundtypen zu unterscheiden: Anwendung auf klassische Situationen, etwa Parallelisierung, im Gegensatz zu transklassischen Situationen (Ambiguität, Polysemy, Komplexität).

Das später diskutierte Beispiel der Berechnung der Fibonacci Zahlen, bezieht sich auf den Topos *Numbers* und den Style *Functional Programming*. Entsprechendes kann für die Objekt-orientierte Programmierung vorgeführt werden, wie dies insb. für ein *Dynamic Semantic Web* von Wichtigkeit ist (Heterarchisierung von Ontologien und Datenbanken).

Anmerkung: botton-up/top-down

Es lässt sich nicht leicht ein reiner botton-up-Standpunkt für eine Präsentation einnehmen, da das System *ConTeXtures* selbst in grosser Allgemeinheit konzipiert ist. Botton-up-Beispiele sind verständlich nur auf der Basis des Gesamtentwurfs entsprechend der allg. Struktur von ARS.

3.2 Aufbauschema der ConTeXture

Das allgemeine Aufbauschema der ConTeXtures hat als "*header*" die globale heterarchische Organisationsstruktur der Verteilung und Vermittlung der lokalen ARS-Systeme, die im "*body*" des Aufbaus erscheinen. D.h., eine gewählte *Heterarchie* als Horizont und Architektonik strukturiert die disseminierten *Hierarchien*. Diese wiederum enthalten die Angaben über Weisen der Programmierung (Styles) und Datentypen (Topics), während der header die Strukturen der Interaktivität und Reflektionalität einer entworfenen Komplexität organisiert, die wiederum durch ihre *categories*, *patterns* und *templates* enthaltend, konkretisiert wird.

ConTeXtures

$$\left[\begin{array}{l} sketch-horizon^{(m)} \\ \left[\begin{array}{l} build-architectonics \\ \left[\begin{array}{l} thematize-scenarios \\ \left[\begin{array}{l} choose-styles \\ \left[\left\{\begin{array}{l} functional \\ imperative \\ object-oriented \\ logical \\ contextural \end{array}\right\}\right] \\ \left[\begin{array}{l} select-topics \\ \left[\left\{\begin{array}{l} Boolean \\ numeric \\ symbolic \\ class \\ relational \\ reflectional \\ actional \end{array}\right\}\right] \\ \left[\begin{array}{l} identify-contextures \\ \left(\begin{array}{l} define-operations \\ \left(\begin{array}{l} abstract-function \\ \left(\left\{propose-statements\right\}\right) \end{array}\right) \end{array}\right) \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

L

Lokalisiert werden die ARS Systeme durch i*dentify contexture,* ihr Apparat wird mit *define*, *abstract* bzw. lambda und den *statements* bestimmt.

$$\left[\begin{array}{l} sketch-horizon^{(m)} \\ \left[\begin{array}{l} build-architectures \\ \left[\begin{array}{l} thematize-szenario \\ \left[\begin{array}{l} identify-contextures \\ \left(\begin{array}{l} define-operations \\ \left(\begin{array}{l} abstract-function \\ \left(\left\{propose-statements\right\}\right) \end{array}\right) \end{array}\right) \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

Operatoren-Schema:

samba (architectonics (reflectionality (interactivity (define (lambda (statements)))))

3.3 Darstellungsmethoden: Matrix, Diagramm und Klammern

Distribution und Vermittlung von vollständigen ARS Programmier-Systemen in der polykontexturalen Matrix. Nicht belegte Stellen der Matrix werden mit einem Leerzeichen, etwa #, versehen, oder aber, aus stilistischen Gründen, nicht notiert.

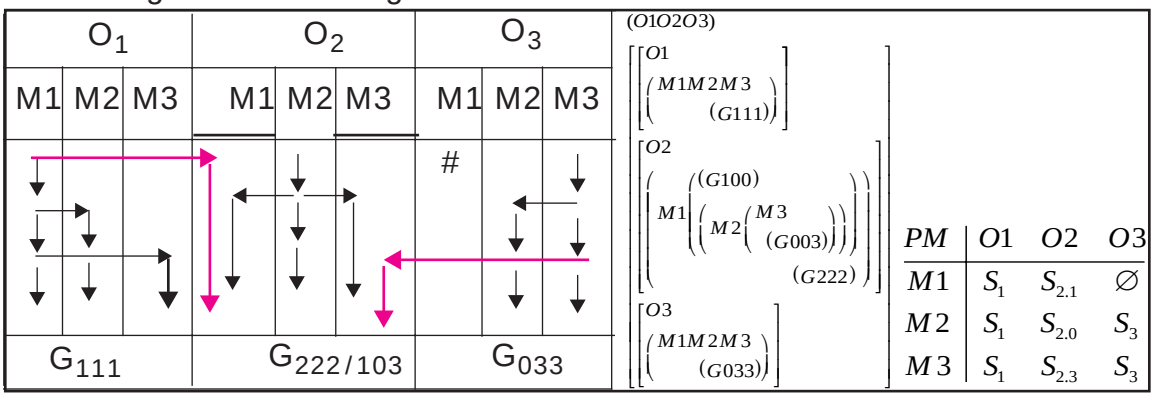
Diagramme visualisieren die Verteilung der Algorithmen und deren Interativität und Reflektionalität. Diagramme haben wenig operative Aussagekraft und werden schnell unübersichtlich.

Die *Klammerdarstellung* lässt sich als Formalismus verstehen, der die oben genannten Eigenschaften abbildet und deren iterative Muster übersichtlich darzustellen in der Lage ist.

Die *Matrix* bietet eine andere Visualisierung an, die auch als Formalismus dienen kann.

Alle drei Formen reflektieren die grundsätzliche Zweidimensionalität, gebildet, hier, durch Interaktivität und Reflektionalität der disseminierten Algorithmensystemen. Damit wird schon auf architektonischer Ebene der der semiotische Zeilenzwang (Max Bense), seine Linearität und Atomizität der Zeichen logozentrischer Formalismen definitiv zu Gunsten einer graphematischen Tabularität der Inskriptionen verlassen.

Diagramm-Darstellung Klammer-Kalkül Matrix



ConTeXtures = DISS (ARS)

DISS = Distribution + Mediation

ConTeXtures =
Computation +
Reflection +
Interaction +
Intervention +
Anticipation

4 Proto-typische Applikationen in ConTeXtures

Was in grosser Allgemeinheit konzeptionell in DERRIDA'S MACHINES und insb. in *Dynamic Semantic Web* skizziert wurde, lässt sich nun im Sprachrahmen der ConTeXtures schrittweise realisieren. Hier beschränke ich die Darstellung auf das proto-typische Beispiel der Berechnung der Fibonacci-Zahlen im Style der funktionalen Programmierung.

4.1 Parallelitäten für Funktionales Programmieren

4.1.1 Strategien der Implementierung

Ein Modell wird eingeführt.

Ein Konstrukt wird vorgeschlagen.

Eine Realisation wird erprobt.

Eine Komparation zu anderen Ansätzen wird vorgenommen.

Beispiel seien die **Fibonacci Zahlen**.

Etappen der Implementierung:

Fibonacci, allgemein

Fibonacci ARS

Fibonacci par (GHC)

Fibonacci poly-ARS (ConTeXtures)

4.1.2 Fibonacci, allgemein

Die Fibonacci Zahlen werden üblicherweise wie folgt definiert:

```
fib: int --> int
fib(1) = fib(2) = 1
fib(n) = fib(n-1)+fib(n-2),  if n>2
```

Dies wird durch eine doppelte Rekursion erreicht. In Java liest sich das als:

```
function fib(n)     // JavaScript
 { if( n <= 2 ) return 1;
   else return fib(n-1)+fib(n-2);
 }

Fibonacci Zahlen: 0, 1, 1 , 2, 3, 5, 8, 13, 21, 34, 55, ...
Etwa für n=20 : 6795
```

> "The corresponding recursive function is a **standard benchmark** for measuring the efficiency of compiled code! It is far to slow for any other use because it computes subproblems repeatetly." Paulson, p.47

4.1.3 Fibonacci in ARS

Das Programm für Fibonacci-Zahlen wird in ARS wie folgt eingeführt. Es wird dabei schon auf sehr elementarer Ebene von der Rekursion, definiert durch den Y-Operator, gebrauch gemacht. Die Definition des Y-Operators wird explizit mit angegeben.

```
(define Y
  (lambda (f)
    ((lambda (x) (f (x x)))
     (lambda (x) (f (x x))))))

(define FIB (Y (fib)))

(ndispl! FIB (five))
```

```
(define fib
  (lambda (n)
    (if (equaln zero)
        (equal one)
        one
        (add (fib (sub n one))
             (fib (sub n two))
             one))))
```

Eine weitere *Analyse*, die die Kontrollstruktur betont und Kantorovic Bäume benutzt, zeigt die folgende Darstellung in einer normierten Notation. Kantorovic-Bäume zeigen sehr schön die *Tiefe* des Algorithmus.

```
fib ≡ ( nat n ) nat :
    if n ≤ 1 then 1
            else fib(n – 2) + fib (n –1 ) fi
```

```
fib ≡ [if . then . else . fi ]
       |      |      |     |
       n     .≤.     1    .+.
             /\            /\
            n  1         fib  fib
                          |    |
                         .–.  .–.
                         /\   /\
                        n  2 n  1
```

Die Kontrollstruktur ist if.then.els.fi

Erste Operator-Ebene

Zweite Ebene

Tiefe ist 2 mit (.+., .–.)
Breite ist 1

O (fib) ist exponential.

Georg Loczewski, ARS: **http://www.aplusplus.net/bookonl/**

Algorithmische Rechenzeit

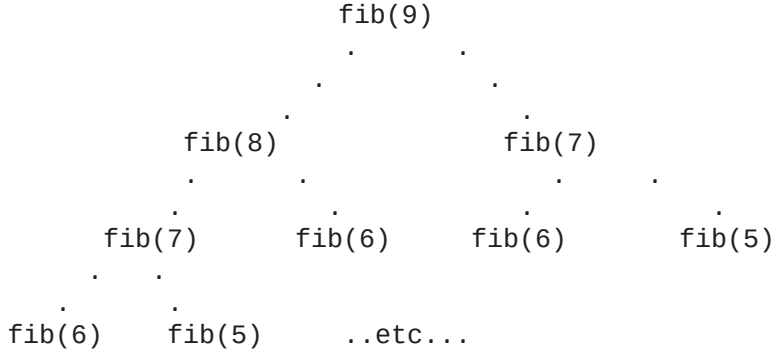
Diese klassische Formulierung des Algorithmus ist korrekt, jedoch wenig effizient wie die Bestimmung der algorithmischen Rechenzeit aufweist. Die Rechenzeit für fib(n) lässt sich durch die Funktion T(n) darstellen.

```
T(1) = T(2) = a            für konstantes a
T(n) = b + T(n-1)+T(n-2)   für konstantes b
     > 2T(n-2)

 Ergebnis, T(n)>a*2n/2.
```

Die Zeit wächst *exponential* mit n.

Der Grund liegt darin, dass Berechnungen, die in einem Zweig schon erfolgt sind, im anderen Zweig wiederholt werden müssen. So ruft etwa fib(9) die Funktionen fib(8) und fib(7) auf. Fib(8) wiederum ruft die schon bekannte Funktionen fib(7) und fib(6) auf.

Diese Iteration lässt sich als Aufruf-Baum darstellen



10.3.1 fibonacci Function, again

```
(define fibonacci
  (lambda (n)
    (if (< n 2)
      n
      (+
        (fibonacci (- n 1))
        (fibonacci (- n 2))))))
```

had an execution time of (+ (* e (expt q n)) (* f (expt r n))) for suitably chosen e, q, f and r.

This time is **exponential** which is clearly greater than any polynomial in n. Hence, the fibonacci function above is an example of an **intractable** computation.

http://www.cs.trinity.edu/About/The_Courses/cs301/10.program.execution.time/10.prog.ex.time.html

4.1.4 Strategien der Optimierung

Die genannte Ineffektivität der Implementierung der Fibnacci-Reihe lässt sich nun durch zwei klassische Strategien korrigieren.

Eine dritte Strategie wird von ConTeXtures angeboten.

Die *erste* klassische Strategie führt *höher-stellige* Funktionen ein, die eine effizientere Berechnung erlauben. Deren Konstrukte sind jedoch, da sie Paare von Funktionen berechnen, nicht mehr so elementar wie die der Ursprungsimplementierung. Dies erzeugt Not gedrungenermassen Kosten an einer anderen Stelle. Diese Strategie verschiebt dabei das Problem auf eine andere Ebene der Hierarchie der Funktionalität und ist nur am *Resultat* und nicht der Strategie der Implementierung interessiert.

Die *zweite* Strategie akzeptiert die funktionale Ausgangssituation und bietet eine klassische Parallelrechnung der gegebenen elementaren Funktionen an. Dafür müssen allerdings neue Konstrukte zur Organisation der Berechnung eingeführt werden: *par* und *seq.* Der Parallelismus basiert auf der algorithmischen Parallelität der Funktionen und ist abgesichert durch das Graph-Reduktionsschema der Kombinatorischen Logik und der Funktionalen Programmierung (Graph-Reduktions-Schema). Programmiertechnisch wird diese zweite Strategie von Wolfgang Loidl auf die Formel *Parallelismus = Algorithmius + Strategie* gebracht.

Erste Optimierungs-Strategie: Paarfunktionen

Eine typische Formulierung für die erste Optimierungsstrategie, basierend auf *nextfib* $(F_{n-1}, F_n) = (F_n, F_{n+1})$, ist wie folgt gegeben.

> The nth Fibonacci number depends on the (n-1)th and (n-2)th numbers. A routine is written to calculate the nth and the (n-1)th numbers from the (n-1)th and (n-2)th numbers. This forms the basis of a simple *linear-recursion.*

```
f: int x int --> int x int
function f(a, b, n)
 { if(n <= 1) return b;
   else return f(b, a+b, n-1);
 }

fib: int --> int x int
function fib(n)
 { return f(0, 1, n); }
```

> Note the two parameters a and b which hold two successive Fibonacci numbers. This *linear* recursive version takes O(n) time.

http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Recn/Binary/

> *"One should not conclude from this that* ***tree-recursive*** *processes are useless. When we consider processes that operate on hierarchically structured data rather than numbers, we will find that tree recursion is a natural and powerfull tool. But even in numerical operations, tree-recursive processes can be useful in helping us to understand and design programs."* Abelson/Sussman, p. 37

4.2 Algorithmischer Parallelismus = Algorithmus + Strategie

Die *zweite* Strategie, die algorithmischer Parallelisierung der Fibonacci Zahlen geht einen Schritt weiter in der Optimierung der Berechnung der Fibonacci Zahlen und wurde auf dem Glasgow Haskell Parallel Copiler (GHC) implementiert.

```
parfib :: Int –> Int
parfib 0 = 1
parfib 1 = 1
parfib n = nf2 `par` (nf1 `par` (nf1+nf2+1))
        where nf1 = parfib (n-1)
              nf2 = parfib (n-2)
```

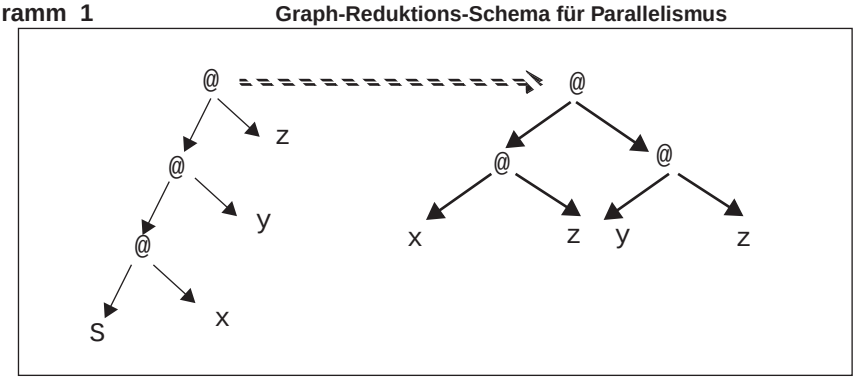
The drawback of this program is the blocking of the main task on the two created child-tasks. Only when both child tasks have returned a result, the main task can continue. It is more efficient to have one of the recursive calls computed by the main task and to *spark* only one parallel process for the other recursive call. In order to guarantee that the main expression is evaluated in the right order (i.e. without blocking the main task on the child task) the *seq* annotation is used:

```
parfib :: Int –> Int
parfib 0 = 1
parfib 1 = 1
parfib n = nf2 `par` (nf1 `seq` (nf1+nf2+1))
        where nf1 = parfib (n-1)
              nf2 = parfib (n-2)
```

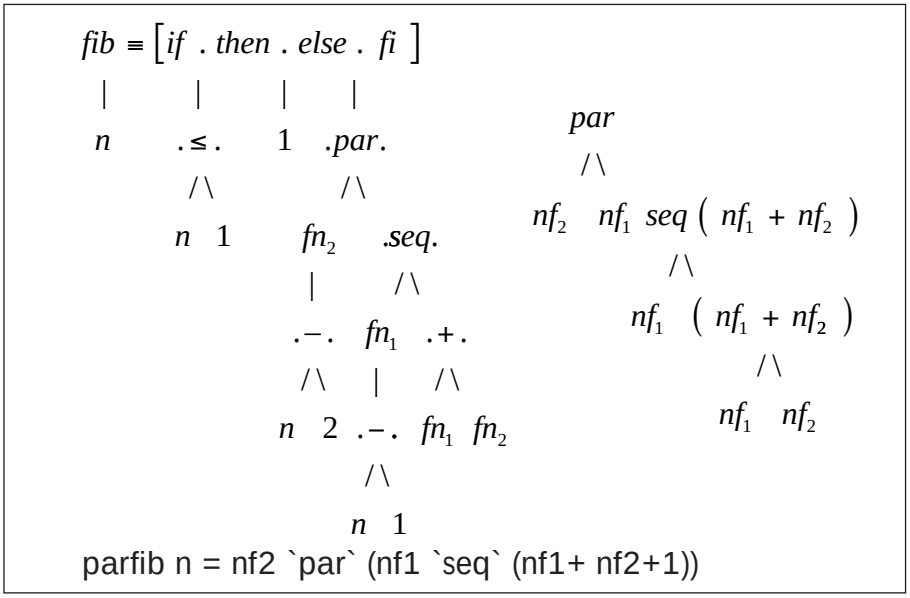
The operational reading of this function is as follows: First spark a parallel process for computing the expression parfib (n-2). Then evaluate the expression parfib (n-1). Finally, evaluate the main expression. If the parallel process has not finished by this time the main task will be blocked on nf2. Otherwise it will just pick up the result.

http://www.dcs.gla.ac.uk/fp/software/gransim/user_2.html#SEC6

Diagramm 1 **Graph-Reduktions-Schema für Parallelismus**



4.2.1 Kantorovic-Analyse der GHC Lösung



Hier gilt der Slogan: *Parallelismus = Algorithmus + Strategie*.

Analyse

Tiefe der Operatoren ist 4 mit 4 Operatoren (.par., .seq., .+., .–.), dabei sind 2 strategische und 2 algorithmische Operatoren beteiligt.

Die Breite ist 1. Das Ganze ist in einer einzigen Kontextur realisiert.

4.2.2 Zum systematischen Ort der Parallel-Operatoren in HASKELL

Die organisatorischen Operatoren "*par*" und "*seq*" sind Konstrukte der funktionalen Programmiersprache HASKELL. D.h., sie sind als korrekte *Erweiterungen* des Sprachrahmens von HASKELL zu verstehen. Sie sind somit nicht genuine Operatoren der Sprache HASKELL selbst, sondern als neue Konzepte und Operatoren in ihr definiert.

Damit entsteht das Problem der prinzipiellen *Reduzierbarkeit* der organisatorischen Operatoren. Konzeptionell lassen sie sich auf die nicht-parallel Grundoperatoren von HASKELL reduzieren. Sie haben damit einzig *pragmatische* und keine konzeptionelle Bedeutung. Diese Anmerkung gilt ebenso für andere nicht-genuin parallele Sprachen , wie etwa SATIN als Erweiterung von JAVA. Nicht jedoch für die spezifischen Parallelsprachen, wie etwa OCCAM. Für allle gilt allerdings, dass sie im Rahmen einer einheitlichen Programmiersprache definiert sind, die als solche nicht distribuiert und damit mono-kontextural definiert ist.

Die Operatoren für Interaktivität und Reflektionalität sind dagegen in ConTeXtures fundamental eingeführt, und sind als solche nicht durch Reduktion zu eliminieren. Mehr noch, jedes distribuierte ARS System kann zusätzlich diese Operatoren für immanente Parallelität enthalten.

4.3 Wer HASKELL nicht mag, kann es auch in SATIN haben

The Satin Programming Model

- **SPAWN and SYNC**
  - Spawned methods *may* run in parallel
  - The sync statement waits until all spawned jobs in *this* method invocation have completed
- **No shared memory**
  - Communication through parameters and return values
  - Replicated objects

4

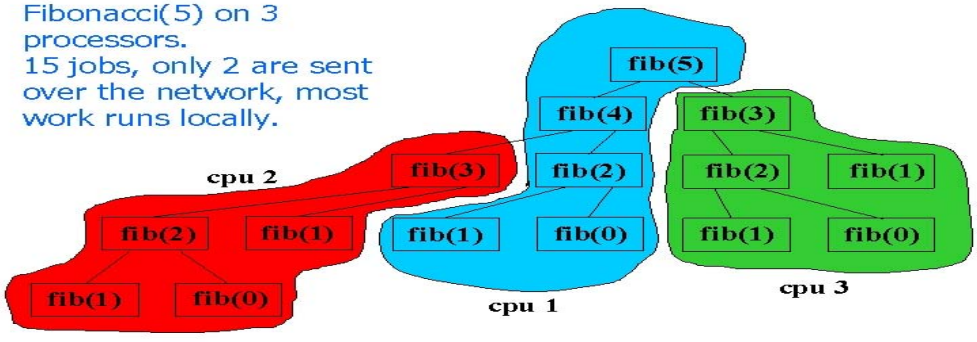
An Example: Parallel Fibonacci (1)

```
SATIN int fib (int n) {
    if (n < 2) return n;

    int x = SPAWN fib (n - 1);
    int y = SPAWN fib (n - 2);
    SYNC;

    return x + y;
}
```

5

An Example: Parallel Fibonacci (2)

Fibonacci(5) on 3 processors.
15 jobs, only 2 are sent over the network, most work runs locally.



6

http://www.cs.vu.nl/~robn/talks/ppopp01/sld005.htm

4.4 Basic Concepts of Parallelity in pLISP

Eine interessante Lösung bietet das pLISP von Thomas Mahler. Es ist zu lokalisieren zwischen den klassischen Ansätzen zur Parallelität und dem hier vorgeschlagenen. Der Unterschied zwischen pLISP und ConTeXtures liegt darin begründet, dass pLISP auf der Basis distribuierter strikter Operatoren mithilfe des neu eingeführten Kombinators P (Proemialkombinator) einen Parallelismus einführt, wohingegen ConTeXtures eine Dissemination auch der nicht-strikten Operatoren (I, K, S) versuchen.

© '98 Thomas Mahler Basic Concepts of Parallelity in pLisp

This File presents some demos for the parallel features of the underlying VM of pLISP

1. strict operations

There are implicit mechanisms that spawn new subtasks if the VM detects possible parallelity. All the standard Combinators S, K, I, Y, B, B', B*, C, and C'are nonstrict operations that don't evaluate their arguments.

Thus there is no chance for any parallelity in evaluating them. But arithmetic operations like (+ x y) need their arguments in normal form.

Thus x and y have to be complete evaluated before the final addition may be performed. Obviously it is possible to reduce x and y in parallel.

Experiments with the fib function will show this behaviour:

```
// define recursive fibonacci function:
input: (define fib (lambda (n) (if (= n 0) 1 (if (= n 1) 1 (+ (fib (-1 n)) (fib (- n 2)))))))
value: (lambda (n) (if (= n 0) 1 (if (= n 1) 1 (+ (fib (-1 n)) (fib (- n 2))))))
// compile it to a combinator logic subroutine (marked by ccsubr)
input: (cf 'fib)
value: (ccsubr_ (S (C (B . if) (C . =) . 0) . 1) (S (C (B . if) (C . =) . 1) . 1) (S ((B* . +) . fib) . -
1) (B . fib) (C . -) . 2)
// compute fib 10:
input: (fib 10)
steps: 3969 :: threads: 6
value: 89
// Read: Combinator VM needed 3969 reductions and spawned 6 parallel threads

// now we enable the stochastic scheduler for full parallel reduction:
input: (stochastic)
value: 1
// now we run fib again
input: (fib 10)
steps: 3969 :: threads: 55
value: 89
```

This means that (fib 10) evaluates to 98, takes 3969 VM steps and runs up to 50 parallel tasks

3. Explicit Parallelity with the P-Combinator

The VM contains a special P Combinator for Parallel reductions.

P x y reduces to QUOTE x y while x and y are running as independend Tasks

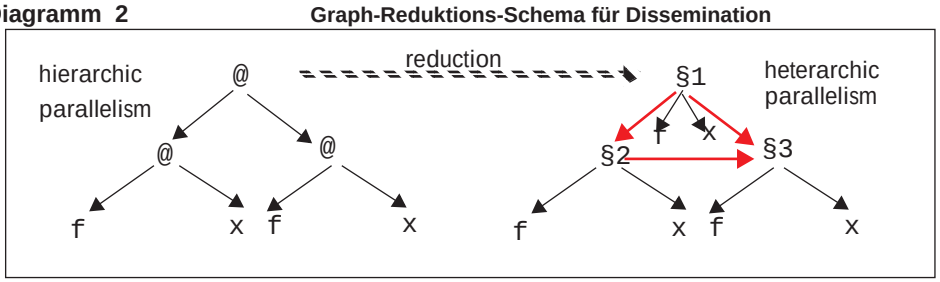
```
(define ptest (lambda (foo)
  (P (+ 1 1) (+ 3 4))))
(cf 'ptest)
(ptest 'bar)
```

http://www.thinkartlab.com/pkl/tm/plisp/linux/par.html

4.5 Architektonischer Parallelismus = Algorithmus + Dissemination

Als dritte Strategie wird eine Modellierung und Programmierung der Fibonacci Zahlen in ConTeXtures vorgenommen. Dabei wird auf die Graph-Reduktion, wie bei der zweiten Strategie der Parallelisierung geschehen, zurückgegriffen. Es könnte allerdings auch die erste Strategie, basierend auf einer Paarbildung, ins Spiel gebracht werden.

Diagramm 2 **Graph-Reduktions-Schema für Dissemination**




```
@ ( @ ( f x ), @ ( f x )) ===> [§1( f x ), §2( f x ), §3( f x )]
```

4.5.1 Modellierungsstrategie

Abbildungen:

nf1 wird in Kontextur1 als fib_1,

nf2 wird in Kontextur2 als fib_2 abgebildet und

die Addition *add* wird in Kontextur3 als fib_3 = (fib1 + fib2) realisiert.

Wie wird modelliert und wie wird das Modell gelesen?

Global —> lokal —> global

$$numeric_0 \rightarrow \left\{ \begin{array}{l} num_1 : int \rightarrow int \\ num_2 : int \rightarrow int \\ num_3 : int \rightarrow int \end{array} \right\} \rightarrow numeric_0$$

$$fib_0 \rightarrow \left\{ \begin{array}{l} fib_1 : int \rightarrow int \\ fib_2 : int \rightarrow int \\ fib_3 : int \rightarrow int \end{array} \right\} \rightarrow fib_0$$

$$with : fib_3 \Rightarrow fib_0$$

Dabei entspricht fib_0 der klassischen, globalen Definition, während fib_i, i=1, 2, 3 die lokalen Funktionen darstellen, die in der *Magic Box* die dekomponierte Funktion berechnen und deren Ergebniss, intern dargestellt durch die lokale Funktion fib3, an die globale Funktion fib_0 weiter gegeben wird.

Die organisatorischen Operatoren *par* und *seq* entfallen vorerst, da die Polykontexturalität des Entwurfs diese Organisationsform schon vorgibt. Erst bei der, über die Programmierung hinausgehenden Implementierung, werden interaktionale und reflektionale Operatoren, Elektoren, als Organisatoren ins Spiel gebracht.

Diagramm-Darstellung

O_1			O_2			O_3		
M1	M2	M3	M1	M2	M3	M1	M2	M3
fib_1 ↓ ↓	#	#	#	fib_2 ↓ ↓	#	#	#	fib_3 ↓ ↓
G_{100}			G_{020}			G_{003}		

Klammer-Kalkül

$$(O1O2O3)\ \left[\left[O1\ \left(\begin{matrix}M1M2M3\\(G100)\end{matrix}\right)\right]\ \left[O2\ \left(\begin{matrix}M1M2M3\\(G020)\end{matrix}\right)\right]\ \left[O3\ \left(\begin{matrix}M1M2M3\\(G003)\end{matrix}\right)\right]\right]$$

Matrix

PM	$O1$	$O2$	$O3$
$M1$	S_1	$\varnothing$	$\varnothing$
$M2$	$\varnothing$	S_2	$\varnothing$
$M3$	$\varnothing$	$\varnothing$	S_3

4.5.2 Implementierung in ConTeXtures (poly-ARS)

Die eingeführte Modellierungsstrategie erlaubt nun eine direkte Abbildung der ARS-Module (fib_1, fib_2, fib_3) in die polykontexturale Matrix. Die verteilte Rekursion wird durch den distribuierten Y-Operator (YYY) gewährleistet. Die 3 belegten Orte entsprechen der Hauptdiagonalen der Matrix, deren Leerstellen hier nicht notiert sind.

$$samba^{(3)}\ (\ id, id, id\ ) \qquad ;;< architecture >$$

$$\left[\begin{array}{lll} thematize\ FIB^{(3)} \quad ;;< topics > & & \\ \left[\begin{array}{l} identify\ contexture_1 \\ \left[\begin{array}{l} define\ fib_1 \\ \left(\begin{array}{l} lambda\ (\ n\ ) \\ \left(\begin{array}{l} if\ \left(\begin{array}{l} equaln\ zero \\ equal\ one \end{array}\right) \\ \quad one \\ (\ fib_1\ (\ sub\ n\ one\ )) \end{array}\right) \end{array}\right) \end{array}\right] \end{array}\right] & \left[\begin{array}{l} identify\ contexture_2 \\ \left[\begin{array}{l} define\ fib_2 \\ \left(\begin{array}{l} lambda\ (\ n\ ) \\ \left(\begin{array}{l} if\ \left(\begin{array}{l} equaln\ zero \\ equal\ one \end{array}\right) \\ \quad one \\ (\ fib_2\ (\ sub\ n\ two\ )) \end{array}\right) \end{array}\right) \end{array}\right] \end{array}\right] & \left[\begin{array}{l} identify\ contexture_3 \\ \left[\begin{array}{l} define\ fib_3 \\ \left(\begin{array}{l} lambda\ (\ n\ ) \\ \left(\begin{array}{l} lambda\ (\ add\ ) \\ \left(\begin{array}{l} if\ \left(\begin{array}{l} equaln\ zero \\ equal\ one \end{array}\right) \\ \quad one \\ (\ add\ fib_1\ \ fib_2\ ) \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right] \end{array}\right] \end{array}\right]$$

$$\left(\ define\ FIB\ \left((YYY)\ FIB^{(3)}\right)\ \right)$$

$$\left(\ ndispl!\ (FIB\ \ five)\ \right)$$

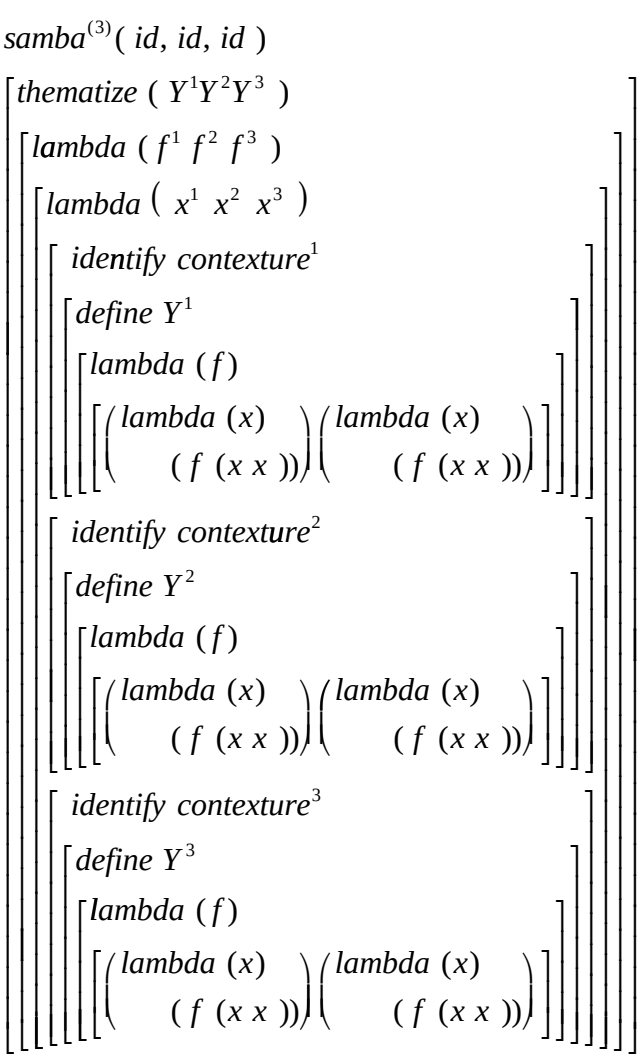
Kantorovic-Bäume für die disseminierte Implementierung von $fib^{(3)}$

```
fib(3) ≡ [[if . then . else . fi ][if . then . else . fi ][if . then . else . fi ]]
 |          |     |      |          |     |      |          |     |      |
 n        .≤.     1    fib1       .≤.     1    fib2       .≤.     1     .+.
           /\            |         /\            |         /\           /\
          n  1          .–.       n  1          .–.       n  1     fib1   fib2
                         /\                      /\
                        n  2                    n  1
```

Wie leicht aus der normierten Darstellung und ihren verteilten Kantorovic-Bäumen ersichtlich, ist die Tiefe der Operatoren 1 und die Breite 3.

Und je Ort, Kontextur, ist die Berechnung linear.

4.5.3 Der Y-Operator als Basis der disseminierten Rekursion



Der disseminierte Y-Operator als Basis der Rekursion der Module wird in ConTeXtures über 3 Kontexturen distribuiert und vermittelt und zwar über der Hauptdiagonalen der Matrix, ohne Interaktivität oder Reflektionalität, sondern in strikter Parallelität, jedoch als Sukzession notiert.

4.5.4 Kommentar

Damit ist ein architektonischer Parallelismus für Fibonacci-Funktionen programmiert. Dass der Parallelismus für Fibonacci garantiert ist, erben wir von den früheren Implementierungen des algorithmischen Parallelismus der Fibonacci Funktion, dargestellt in dem polykontexturalen Graph-Reduktions-Schema. Die Konstrukte als solche sind jeweils in ARS implementiert. Wir haben jedoch noch kein implementiertes und lauffähiges poly-ARS (ConTeXtures) zur Verfügung. Als eine erste Simulation von poly-ARS ist es ratsam, die einzelnen Module isoliert in ARS zu implementieren und entsprechend isoliert auf einem Rechner ablaufen zu lassen. Die Vermittlung zwischen den in der Simulation isolierten Modulen muss dann allerdings nachträglich "von Hand" gemacht werden.

Es sollte dabei trotzdem möglich sein, über den konzeptionellen Gewinn hinaus, auch den praktischen Mehrwert der Konstruktion zu demonstrieren.

4.5.5 Komparation

Es gilt nun zu zeigen, dass der architektonische Parallelismus mit seiner distribuierten Rekursion basierend auf dem disseminierten basalen Y-Operator nicht nur der non-parallelen, sondern auch der algorithmisch-parallelen Implementierung konzeptionell wie auch berechnungstechnisch überlegen ist. Die Rechenzeit für $fib^{(3)}(n^{(3)})$ lässt sich durch die Funktion $T^{(3)}=(n1, n2, n3)$ darstellen. Völlig neu ist dabei die *Breite* der Berechnung, bzw. der algorithmischen Zeit, die zusätzlich zur *Tiefe* der Berechnung zu analysieren ist. Dabei ist die Breite immer linear und abhängig von der Anzahl und Verteilungsstruktur der beteiligten Kontexuren der Berechnung. Aus dem Kantorovic-Baum der disseminierten Implementierung lässt sich direkt ablesen, dass die Zeit lokal linear O(n) und global über 3 Orte verteilt ist, gegensätzlich zum exponentiellen Resultat der ursprünglichen Implementierung.

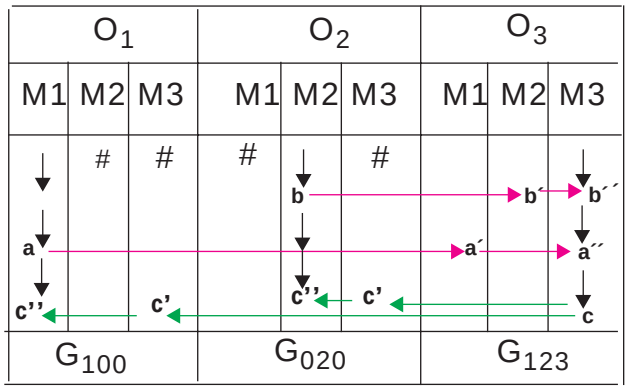
Resultat.Offensichtlich ist die Implementierung der Fibonnacci Reihe in ConTeXtures in verschiedener Hinsicht effektiver als die klassische.

4.5.6 Implementierung = Dissemination + Interaktivität + Reflektionalität

Das Interaktionsmuster zwischen den Teilfunktionen wird in den klassischen Fällen nicht direkt programmiert, da es der Maschine zugeordnet wird. Nichtdestotrotz müssen natürlich die jeweiligen Werte gespeichert, aufgerufen, übergeben und weiter verarbeitet werden.

Im Fall der ConTeXtures lässt sich dieses Interaktionsmuster, nun verteilt über 3 Kontexturen, direkt modellieren und programmieren.

Es ist nicht nur zur Explikation des Modells von Bedeutung, sondern auch für die konkrete Implementierung, dieses Interaktions- und Reflektionsmuster der Kalkulation des Algorithmus explizit anzugeben.



In this example **a´** represents to the machine M3 at the place O_3 the value of processor M1which delivered the value **a** for itself in its own space. The value or object **a´** is semiotically not identical with the object **a** but it is the same, that is, it is a duplication of **a**. This duplication of **a** is not well understood in terms of data processing. It is a transition

from one contexture to another one. Such operations are part of the poly-contextural framework independently of the systems which are localized in it. Therefore the proposed model is nevertheless not a model of data-sharing or similar. Data-sharing and message-passing, surely, is not excluded but it is a secondary feature of the whole system.

The mirrored objects in contexture3 of contexture1 and contexture2 are not *identical* to the objects **a´´** and **b´´** in use by the main task of machine3, they are, again, only the *same*, that is, they are objects used *as* objects for computation by machine3 as they are delivered by the other machines. These objects are having 3 different occurrences in respect to their 3 different roles in the whole game. All these roles are played by the objects more or less simultaneously. As a final result of this interaction, machine3 is delivering the result **c**. The inverse happens, if necessary, machine3 is delivering back to the machine1 and machine2 the obtained result **c** as **c'** for further calculations. This may be programmed with the help of the operator *elect* as shown below.

Die vollständige Implementierung für Fibonacci in ConTeXtures

$$samba^{(3)}\ (\ id, id, id\ )\qquad ;;< architectonics : id-super-operators, complexity = 3 >$$

$$\left[\begin{array}{l} style\ \ functional \\ \left[\begin{array}{l} topics\ \ numeric \qquad ;;<\ fib_i : int_i --> int_i,\ i = 1, 2, 3\ > \\ \left[\begin{array}{l} thematize\ FIB^{(3)} \qquad ;;< scenario : FIB^{(3)} = \left(fib_1,\ fib_2,\ fib_3 \right) \\ \left[\begin{array}{l} identify\ \ contexture_1 \\ \left[\begin{array}{l} define\ fib_1 \\ \left(lambda\ (\ n\ ) \right. \\ \left. \left(\left(\left(if\ \left(\begin{array}{l} equaln\ zero \\ equaln\ one \end{array} \right) \right. \right. \right. \right. \\ \qquad one \\ \left. \left. \left. \left. \left(fib_1\ (\ sub\ n\ one\ ) \right) \right) \right) \right) \right) \end{array} \right] \\ \left(elect_3 \right) \end{array} \right] \left[\begin{array}{l} identify\ \ contexture_2 \\ \left[\begin{array}{l} define\ fib_2 \\ \left(lambda\ (\ n\ ) \right. \\ \left. \left(\left(\left(if\ \left(\begin{array}{l} equaln\ zero \\ equaln\ one \end{array} \right) \right. \right. \right. \right. \\ \qquad one \\ \left. \left. \left. \left. \left(fib_2\ (\ sub\ n\ two\ ) \right) \right) \right) \right) \right) \end{array} \right] \\ \left(elect_3 \right) \end{array} \right] \left[\begin{array}{l} identify\ \ contexture_3 \\ \left[\begin{array}{l} define\ fib_3 \\ \left(lambda\ (\ n\ ) \right. \\ \left(lambda\ (\ add\ ) \right. \\ \left. \left. \left(\left(\left(if\ \left(\begin{array}{l} equaln\ zero \\ equaln\ one \end{array} \right) \right. \right. \right. \right. \right. \\ \qquad one \\ \left. \left. \left. \left. \left. \left(add\ fib_1\ fib_2\ \right) \right) \right) \right) \right) \right) \end{array} \right] \\ \left(elect_1\ elect_2 \right) \end{array} \right] \end{array} \right] \end{array} \right] \end{array} \right]$$

In dieser Implementierung werden die *organisatorische Operatoren* wie etwa (par, seq) *explizit* ins Spiel gebracht, nämlich als *Elektoren*. Elektoren *elect* wählen zwischen Kontexturen, während Selektoren *sel* lokal gelten. Beide haben eine basale Bedeutung, *sel* intra- und *elect* transkontextural. Die Kontrollstruktur [if.then.else.fi] basiert auf dem Selektor *sel*. *Transkontexturale Übergänge* (Günther) werden durch Elektoren realisiert, die in der Lage sind, Kontexturen als Ganze zu evozieren.

5 Bilanz

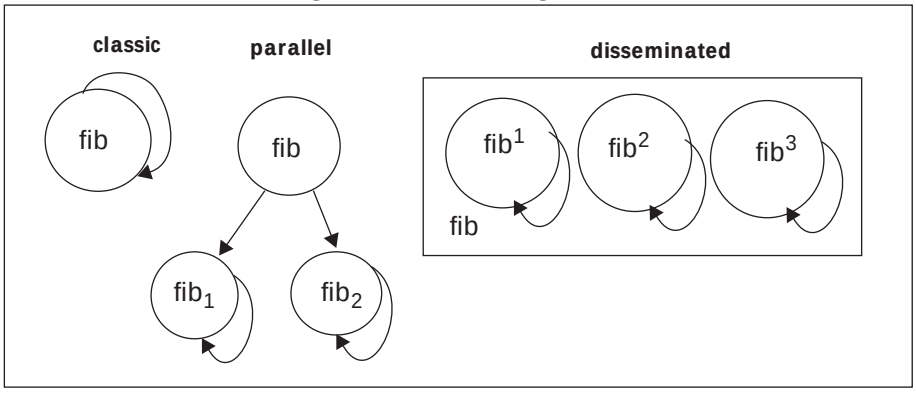
Entsprechend der Unterscheidung von daten-bezogener, algorithmischer (nicht-paralleler und paralleler) und architektonischer Komplexität lassen sich je Realisationstyp verschiedene Optimierungsstrategien und deren Kosten unterscheiden.

Nachteile *der polykontexturalen Modellierung.* Die architektonische Parallelverarbeitung hat verglichen mit allen anderen Modellen einzig den Nachteil, dass sie zu ihrer Realisation konzeptionell von vornherein von einer Dissemination sowohl der Programm-Module wie auch der Prozessoren ausgeht, die die bekannten Ansätze übersteigt. Beide Aspekte erweisen sich jedoch bei genauerer Betrachtung als Vorteile.

Die Programmierung einer Distribution von Modulen ist leichter als das Konstruieren und Pflegen eines grossen und einheitlichen Programms, das zwar auch modular aufgebaut sein kann, dessen Module jedoch einzig *hierarchisch* verteilt werden können. Module in ConTeXtures lassen sich sowohl hierarchisch wie *heterarchisch*, d.h. sowohl vertikal wie horizontal organisieren. Damit ist eine weit grössere Flexibilität der Programmierung gewährleistet als dies in anderen bekannten modularen Programmiersystemen, wie etwa der OOP möglich ist. Des Weiteren ist zu bedenken, dass die Kosten von Hardware-Prozessoren dramatisch sinken und jetzt schon verschwindend klein sind gemessen an den enormen und ständigen Kosten der Entwicklung und Pflege der Programme.

Vorteile sind vorab die völlig neue Features der *Interaktionalität* und *Reflektionalität* zusätzlich zur distribuierten Form der Parallelisierung von Berechnungen. Damit entseht nicht nur eine erhöhte *Performanz,* sondern auch eine Steigerung der Sicherheit der Systeme basierend auf der Erhöhung der Direktheit des Zugriffs in der Interaktion mit der Umgebung für Echt-Zeit-Systeme.

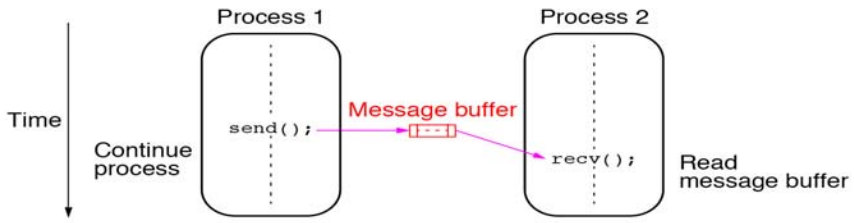
Zusammenfassend lassen sich die 3 Modellierungen der Fibonacci Funktion und ihrer Rekursivität als ein Beispiel der Applikation von ConTeXtures durch die folgenden drei Diagramme darstellen.



6 Nochmals, Interaktivität vs. Kommunikation

How message-passing routines can return before message transfer completed

Message buffer needed between source and destination to hold message:



In der OxM-Matrix sind die jeweiligen nicht zentralen M_is innere Umgebungen des Systems. Diese können verschiedene Funktionen einnehmen, die einfachste ist gewiss als Memory oder Stack für die Daten des Modells des Nachbar-Agenten, zu fungieren. Eine innere Umgebung ist jedoch nicht auf die Speicherfunktion zu reduzieren, da sie ebenso aktive Berechnungsfunktion (algorithmisch wie logisch) haben kann. Die organisatorischen Operatoren, welcher Implementierung auch immer, etwa als (*par, seq*) oder (*spawn, sync*), verlangen allerlei Formen von Repräsentationen, die ausserhalb der eigentlichen algorithmischen und logischen Prozesse lokalisiert sind, etwa als *Buffer*, und daher ihre entsprechenden zusätzlichen Kosten verursachen.

Diese Funktionalität der Organisation wird auf natürliche Weise in ConTeXtures auf der Basis einer durch Reflektionalität und Interaktivität erweiterten Konzeption des Algorithmischen und der Logik realisiert. Einerseits lassen sich die Erfahrungen, Konzeptionen und Methoden der Kommunikations orientierten Implementierung in ConTeXtures übersetzten, andererseits lassen sie sich radikalisieren im Sinne eines Übergangs von einem Kommunikations- zu einem Interaktion-Reflexions-Paradigma der Aktion zwischen autonomen Agenten.

Interaktion basiert auf *Anfragen* und nicht auf Anrufen (send, receive). Anfragen können verworfen (rejiziert) oder angenommen (akzeptiert) werden. Der Angefragte bildet ein inneres Modell des Anfragers, erst wenn ihm dies gelingt, kann er in einen Kommunikationsprozess treten. Im Kommunikationsmodell, definiert durch (process, send, receive, buffer), muss diese basale Begegnungsstruktur der Agenten durch den Designer, zusätzlich zur nicht-interaktiven Struktur der Algorithmen, vorgeben werden.

Logische Äquivalente für Interaktion in ConTeXtures sind die *Transjunktionen.* Transjunktionen spalten sich in zwei verschiedene Teile. In welche, die innerhalb des Systems verbleiben dem sie entstammen und solchen, die in anderen Systemen lokalisiert sein wollen. Ist das Nachbarsystem strukturell nicht in der Lage, einer solche Teilformel eines anderen Systems einen passenden Ort zu ihrer Realisation anzubieten, misslingt die Interaktion. Die Bifurkation der Transjunktion greift ins Leere. Andererseits kann es gelingen, dass die Transjunktion ihren Ort ausserhalb ihres Stammsystems selbst erzeugt. Wird diese Generierung eines Ortes als innerer Umgebung eines Nachbarsystems zur Realisation der Transjunktion von diesem strukturell akzeptiert, ist die Interaktion gelungen. Auf dieser logisch-strukturellen Ebene stehen offensichtlich noch keine Datentypen zur Verfügung, die kommunikativ ausgetauscht werden könnten.