Fraglets: Computing with Macromolecules
A Tutorial

BIONETS SP2, Fraglets Workshop
Brussels, January 29-30, 2007

Lidia Yamamoto, University of Basel
Tutorial Overview

- Introduction: Original Fraglets, Fraglets in BIONETS
- Fraglets Programming Basics: original and new instructions, examples
- Programming Methodology: breaking down complexity, backwards derivation
- Tools: Automatic code generator (partial), concentration and rate plots, reaction graph
- Exercises
- Summary and Discussion
Fraglets: Background

• Creation around 2001 by C. Tschudin [AINS’03]
  – Inspiration: Molecular biology, cell metabolism, chemical computing (Membrane Computing, Gamma, CHAM), multiset rewriting

• Goals:
  – Automated protocol synthesis and evolution
  – Unified code and data representation (active + passive networking)
  – Efficient packet processing engine: simple instructions with constant (short!) processing time
Fraglets: Background

- Resulting language:
  - **Fraglet** = computation fragment = code = data = packet
  - Header tag matching, analogous to packet header processing
  - “Assembly language” of chemical computing: micro-instructions, human-unreadable programs, “write-only” code!
Fraglets in BIONETS

- **On-line evolution**: start from working implementations, continuous self-optimization
  - Resilient execution: resist lost or damaged code portions (resist harmful mutations)

- **Service Evolution**, in addition to Protocol Evolution
  - From a protocol-specific language to a more generic computation model (how generic?)
Fraglets Programming Basics

- Syntax
- Instruction set
- Simple programs
Fraglets Syntax

- Execution environment: multiset of fraglets
  - multiset = unordered set in which elements appear more than once

- Fraglet original syntax: string \( n[s_1 : s_2 : \ldots : s_n]m \)
  - \( n \) = node where fraglet executes
  - \( m \) = multiplicity counter: number of occurrences of fraglet in multiset

- New simplified syntax (Dec’06): \( n[s_1 \ s_2 \ldots \ s_n]m \)
  - ‘:’ now optional

- Goal: simple syntax that can be easily manipulated by automatic means (e.g. genetic programming)
Basic Instruction Set [AINS’03]

Transformation rules: involve a single fraglet

[dup t s tail] --> [t s s tail]  # duplicate symbol
[exch t s1 s2 tail] --> [t s2 s1 tail]  # swap symbol
[split f1 * f2] --> [f1], [f2]  # break at ’*’ position
[nul tail] --> []  # discard fraglet
[a[send ch b tail]] --> b[tail]  # UDP-style send

[new t tail] --> [t n_{i+1} tail]  # new tag creation
                          # (never implemented?)

(!) ’:’ made optional > fraglets v0.23
Reaction rules: involve two fraglets

- **Merge if match:**
  \[
  \text{match} \ t \ \text{tail1}, \ [t \ \text{tail2}] \rightarrow \ [\text{tail1} \ \text{tail2}]
  \]

- **Persistent match ("catalyst"):**
  \[
  \text{matchp} \ t \ \text{tail1}, \ [t \ \text{tail2}] \rightarrow \ [\text{matchp} \ t \ \text{tail1}], \ [\text{tail1} \ \text{tail2}]
  \]

- **Sustain variant:**
  \[
  \text{matches} \ s \ t \ \text{tail1}, \ [s \ t \ \text{tail2}] \rightarrow \ [\text{tail1} \ \text{tail2}], \ [s \ t \ \text{tail2}]
  \]
  
  # (never implemented?)
Other instructions

Fraglets v. 0.18 and 0.19, 2005 (www.fraglets.net)

Instructions added after AINS’03 (e.g. WAC’05) or non-documented:

[nop tail] --> [tail]    # nop: do nothing

[wait tail] --> ...       # wait: delayed execution
    # (after 10 execution steps:)
... --> [tail]

[pop t x tail] --> [t tail]    # pop: delete symbol
Simple Programs

- Rewrite header tag
- Append symbol
- Code Mobility
- Lossy link emulation (WAC’05)
Rewrite (rename) header tag

goal: \[\text{in tail} \] --> \[\text{out tail} \]
solution: \[\text{match in out}, \text{in tail} \] --> \[\text{out tail} \]

Append (constant) symbol

goal: \[\text{in tail} \] --> \[\text{out tail s} \]
predefined: \[\text{store s} \]
solution: \[\text{match in match store out} \]
trace: \[\text{match in match store out}, \text{in tail} \] --> \\
\[\text{match store out tail}, \text{store s} \] --> \\
\[\text{out tail s} \]
Code Mobility

Read temperature at remote node

*a a ch*
*a b ch*
*f a[send ch b match temp send ch a tempis]*

*f b[temp 30]*

Result:

*f a[tempis 30]*
50% loss on average:

a a ch
a b ch
f a[transmit b msg]100
f a[matchp transmit send ch]
f a[matchp transmit nul]

Result:

f b[msg]44
Lossy Link Emulation

25% loss on average:

a a ch
a b ch
f a[transmit b msg]100
f a[matchp transmit send ch]3
f a[matchp transmit nul]

Result:

f b[msg]74

Easy to emulate other loss patterns, delays \(\text{nop, wait}\), etc.
New instructions (Dec. 2006, Experimental!)

Basic number manipulation: (currently positive integers only)

[length t tail] --> [t len tail] # length in symbols
[sum t n m tail] --> [t n+m tail] # sum two numbers
[lt yes no n m tail] --> # less than:
    if n < m then [yes n m tail] # compare two numbers
    else [no n m tail]

Examples:

[length t a b c] --> [t 3 a b c]
[sum total 3 4 rest] --> [total 7 rest]
[lt y n 1 2 rest] --> [y 1 2 rest]
[lt y n 9 7 rest] --> [n 9 7 rest]
New instructions (Dec. 2006, Experimental!)

-copy tail --> [tail]2 # copy fraglet

-empty y n tail --> # test if tail empty
  if tail==[] then [y] # (useful for recursion)
  else [n tail]

Examples:

-copy this is a fraglet] --> [this is a fraglet]2

-empty finish continue 6 7 8] --> [continue 6 7 8]
[empty finish continue] --> [finish]
New instructions (Dec. 2006, Experimental!)

# create new symbol:
[newname t s1 s2 tail] --> [t s1s2 tail]

# create new node with communication channel:
[newnode ch node tail] --> a node ch, node[tail]

Examples:

[newname t myid 10 rest of fr] --> [t myid10 rest of fr]
[newnode ch b init b mycode] --> a b ch, b[init b mycode]
More Programs

- Increment counter
- Prepend, append
- Delete from head (reimplement pop)
Increment counter

goal: \([\text{incr x n}] \rightarrow [x \ n+1]\)

How to program: derive code from bottom to top:

\([\text{matchp incr ........... }], [\text{incr x n}] \rightarrow\)

\[
\ldots
\]

\[
\ldots
\]

\[
[\text{sum x 1 n}] \rightarrow \# \text{ step1: find rule that uniquely leads to}
\]

\[
[x \ n+1] \quad \# \text{ target result: this rule is: sum 1 to n}
\]

Resulting program:

\(f \ [\text{matchp incr ........... }]\)
Increment counter

goal: \([\text{incr } x \ n] \rightarrow [x \ n+1]\)

How to program: derive code from bottom to top:

\([\text{matchp incr \ldots \ldots \ldots}]\), \([\text{incr } x \ n] \rightarrow\)

\(\ldots\)

\([\text{exch sum } 1 \ x \ n] \rightarrow \# \text{ step2: find rule that uniquely leads}\)

\(\quad \# \text{ to step1, while pushing input as close as}\)

\(\quad \# \text{ possible to tail: 'exch' does the job!}\)

\([\text{sum } x \ 1 \ n] \rightarrow \# \text{ step1: find rule that uniquely leads to}\)

\([x \ n+1] \quad \# \text{ target result: this rule is: sum } 1 \text{ to } n\)

Resulting program:

\(f \ [\text{matchp incr \ldots \ldots \ldots}]\)
Increment counter

goal: \([\text{incr } x \ n] \rightarrow [x \ n+1]\)

How to program: derive code from bottom to top:

\([\text{matchp incr exch sum 1}], \ [\text{incr } x \ n] \rightarrow \ # \ \text{step3: input is}
\# \ \text{now at tail, so just match header tag and done!}
[\text{exch sum 1 x n}] \rightarrow \ # \ \text{step2: find rule that uniquely leads}
\# \ \text{to step1, while pushing input as close as}
\# \ \text{possible to tail: 'exch' does the job!}
[\text{sum x 1 n}] \rightarrow \ # \ \text{step1: find rule that uniquely leads to}
[x \ n+1] \ # \ \text{target result: this rule is: sum 1 to n}

Resulting program:

\(f \ [\text{matchp incr exch sum 1}]\)
Prepend fraglet

goal: [store 7 8], [prepend 4 5 6] --> [store 4 5 6 7 8]

Trace (code derived from bottom to top):

[matchp prepend match store store], [prepend 4 5 6] -->
[match store store 4 5 6], [store 7 8] -->
[store 4 5 6 7 8]

Resulting program:

f [matchp prepend match store store]
Append fraglet

goal: [store 1 2], [append 3 4 5] --> [store 1 2 3 4 5]

Trace (read bottom-up):

[matchp append split match store match app1 store * app1], [append 3 4 5] -->
[split match store match app1 store * app1 3 4 5] -->
[match store match app1 store], [app1 3 4 5]
[match store match app1 store], [store 1 2] -->
[match app1 store 1 2], [app1 3 4 5] -->
[store 1 2 3 4 5]

Resulting program:

f [matchp append split match store match app1 store * app1]
Delete from head (pop)

Goal: reimplement pop instruction (call it 'del')

[del tag x tail] --> [tag tail]

Trace (backwards derivation, i.e. read bottom-up):

[matchp del exch tmp2], [del tag x y z] -->
[matchp tmp2 exch tmp1 * ], [tmp2 x tag y z] -->
[matchp tmp1 split nul], [tmp1 x * tag y z] -->
[split nul x * tag y z] --> [nul x], [tag y z]

Program:

f [matchp del exch tmp2]
f [matchp tmp2 exch tmp1 * ]
f [matchp tmp1 split nul]
Recursion

Count fraglet length (without using “length” rule):

[count a b c] --> [total 3] # consumes original fraglet

Resulting program:

f [counter 0]
f [matchp count empty stop cnt]
f [matchp stop match counter total]
f [matchp cnt pop cnt1]
f [matchp cnt1 split match counter incr counter * count]
Recursion

Trace: (!) here forward (top-down) fine!

[counter 0]
[matchp count empty stop cnt], [count] --> [total 0]
[matchp count empty stop cnt], [count x tail] -->
  [match counter incr counter], [count tail]
[matchp stop match counter total], [stop] -->
  [match counter total], [counter n] --> [total n]
[matchp cnt pop cnt2], [cnt x tail] -->
  [pop cnt2 x tail] --> [cnt2 tail]
[matchp cnt2 split match counter incr counter * count],
  [cnt2 tail] -->
[split match counter incr counter * count tail] -->
  [match counter incr counter], [count tail] #recursion
Programming Methodology

Break down complexity:

- Identify partial goals: write them down in terms of transformations of the form:
  \[ \text{[intag ...]}, \text{[...]} \rightarrow \text{[outtag ...]}, \text{[...]} \]

- Recursion = reuse of partial goals (good!)

- Solve each partial goal using bottom-up derivation (parts of it can be automated, see following slides)

- In case of manual derivation, keep traces for future use (because resulting program is generally unreadable!!)
Programming Methodology

- Beware of parallel execution: is your code reentrant?
- Test and debug each partial goal separately
- Test full program: can only work!
Tools

- Automatic Code Generator (partial): `gencode.pl`
- Concentration plot: `concentr.pl`
- Production/Consumption rate: `rate.pl`
- Reaction graph: `log2graph*`
Automated Code Generation with gencode.pl

Goal:

input: [tag x tail]
output:
  [frag with x here another x there and again an x tail]

Invoke gencode.pl script:

bin/gencode.pl

 tag x
  f [frag with x here another x there and again an x]
<CTRL-D>
Automated Code Generation with gencode.pl

Output program:

f [ matchp tag dup tag_8 ]
f [ matchp tag_8 exch tag_7 an ]
f [ matchp tag_7 exch tag_6 again ]
f [ matchp tag_6 exch tag_5 and ]
f [ matchp tag_5 exch tag_4 there ]
f [ matchp tag_4 dup tag_3 ]
f [ matchp tag_3 exch tag_2 another ]
f [ matchp tag_2 exch tag_1 here ]
f [ matchp tag_1 frag with ]
Automated Code Generation with gencode.pl

Execution:

```
f [matchp ...] # paste automatically generated code here
f [tag mysymb rest of fra] # example input
f [tag yoursymb second test] # another example input
e # execute
```

Result:

```
f [matchp ...] # same matchp rules, omitted
f [frag with mysymb here another mysymb there and again
    an mysymb rest of fra]1
f [frag with yoursymb here another yoursymb there and again
    an yoursymb second test]1
```
Automated Code Generation

- Able to transform an input symbol into any arbitrary output fraglet
- Saves tedious symbol manipulations
- Deterministic code generation, 100% correct by construction (except for bugs in the generator itself...)
- Simple, but useful feature, since this pattern is very common: for example, in RDP (WAC’05):
  
  \([\text{rdp payload}] \rightarrow [\text{transmit payload}], [\text{store payload}]\)
  
  – transmit one copy of payload to destination and store other copy for retransmission in case of loss
Automated Code Generation

• Recursion supported implicitly (fraglet tail carried along by default)

• Limitations
  – Currently only one input variable supported
  – Implementation maybe not the shortest possible
  – Do we actually need such micro-transformations, or is this rather a language limitation? Why not adding features in the language that allow for any fraglet to be generated with a single rule?

  * Trade-off: complexity of the language vs. complexity of the interpreter
Concentration Plot: concentr.pl
Rate Plot: rate.pl

- Rate of rule1 (fraglets per sim step)
- Time (simulation steps)

production
net rate
consumption
Reaction Graph: log2graph* scripts
Exercises (free choice of one or more)

- Get the minimum of a list of numbers
  \[\text{getmin} \ 8 \ 99 \ 4 \ 23\] \implies \[\text{min} \ 4\]

- Invert a fraglet:
  \[\text{invert} \ a \ b \ c\] \implies \[\text{inverted} \ c \ b \ a\]

- Duplicate a fraglet (without using “copy” rule):
  \[\text{mycopy} \ a \ b \ c\] \implies \[a \ b \ c\]2

- Multiply two numbers:
  \[\text{multiply result} \ x \ y\] \implies \[\text{result} \ x*y\]
Exercises (free choice of one or more)

- Recreate the original ‘new’ instruction using ‘newname’ and ‘sum’ (or ‘incr’):
  
  \[\text{new } t \text{ tail}] \rightarrow \text{[t n}_{i+1}] \text{ tail]} \\

- Mutate a fraglet at a random position, by inserting, deleting or exchanging a symbol, for example:
  
  \[\text{mutate a b c}] \rightarrow \text{[mutated b c]}
  
  \[\text{mutate a b c}] \rightarrow \text{[mutated b a c]}
  
  \[\text{mutate a b c}] \rightarrow \text{[mutated a a b c]}
Installing, Compiling and Running Fraglets

#unpack:
tar xzvf fraglets0.28.tgz

#compile (if needed)
cd src
make fraglets

#run:
./fraglets -d 3 -e 3 -lim 1000 < myprogram.fra

Knoppix CDs available for those without Linux or MacOS.
Solution to Exercise: Getmin

Get the minimum of a list of numbers

Trace:

[getmin n] --> [min2 1 1 n]
[getmin n tail] --> [getmin2 1 len tail]
[min2 1 1 n] --> [min n]
[getmin2 1 len a b rest] --> if a<b [islt a b rest]
else [nlt a b rest]
[islt a b rest] --> [nlt b a rest]
[nlt a b rest] --> [getmin b rest]
Solution to Exercise: Getmin

Program:

f [matchp getmin length len1]
f [matchp len1 lt getmin2 min2 1]
f [matchp min2 pop d1]
f [matchp d1 pop min]
f [matchp getmin2 pop d11]
f [matchp d11 pop getmin3]
f [matchp getmin3 lt islt nlt]
f [matchp nlt pop getmin]
f [matchp islt exch nlt]
Outlook and Perspectives

• Nice programming model, enticing concepts and programs.
• But code is long, complicated and unreadable by humans: “write-only programs”...
• Fully automated deterministic code generation still impossible.
• Can we generate code by other means, e.g. Genetic Programming?
• Should we have a higher-level, human-oriented chemical programming language?
  – if yes, should it be used standalone or compiled into fraglet code?
Outlook and Perspectives

Why do we need a chemical language at all?

- high parallelism: parallel, alternative execution paths: resilient to program transformations in one path, other paths can take over
  - “rerouting” execution flows
  - must still be verified...
  - is there an alternative for on-line software evolution?