On Speaking Languages

Luca Cardelli
Microsoft Research

MacQueen Fest, Chicago, 2012-05-12
http://lucacardelli.name

Outline

- The Language of Science
- The Language of Functions
- The Language of Objects
- The Language of Molecules

The Language of Science

The Language of Science

- Never much good at natural languages.
  - Latin: bad. English: worse.
- Middle-school remedial lessons
  - English is the Language of ~~Shakespeare~~ Science!
  - Learned by Science Fiction, from Asimov to Zelazny.
- In Edinburgh, only two English speakers
  - Robin (textbook, slow English)
    - Lectures on Operational Semantics
  - Dave (American English has 5 vowels instead of 1)
    - Lectures on Hope
  - Corollary
    - Famous episode involving a Scottish milkman and 3 months worth of empty milk bottles stacked all over Gordon Plotkin's flat.

The Language of Functions

The Language of Functions

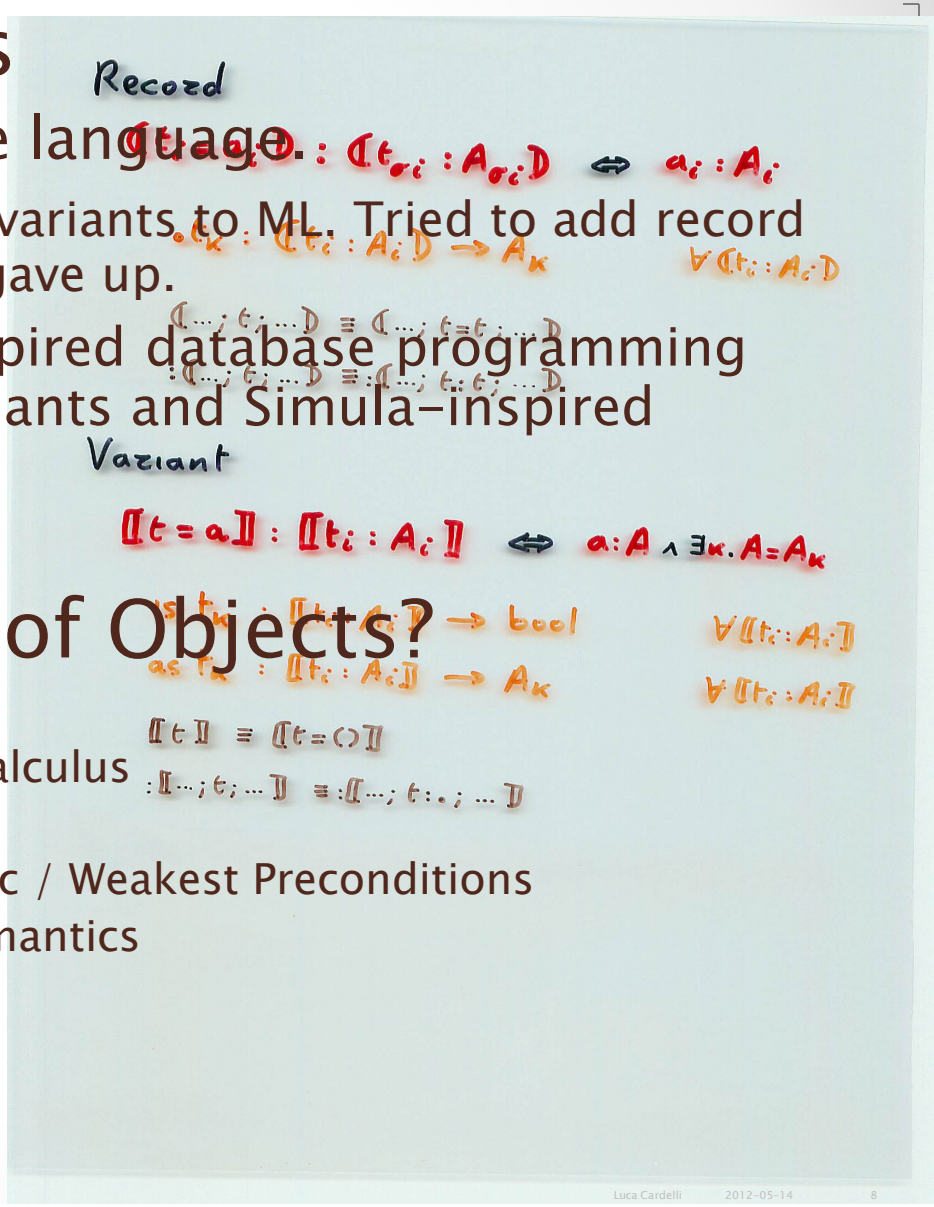
- Ironically attracted to artificial languages
  - Pisa: λ-calculus (+ Fortran, Lisp, Algol68, Simula)
  - Edinburgh: ML & Hope (+ Pascal, VAX assembly)
  - Murray Hill: C and Unix
    (during a snow storm at Dave's)

- And their semantics
  - Pisa: Scott-Strachey semantics.
  - Edinburgh: CPOs.
  - Murray Hill: the MacQueen-Sethi-Plotkin ideal model
    - More on this in a moment…

The Language of Objects

The Language of Objects

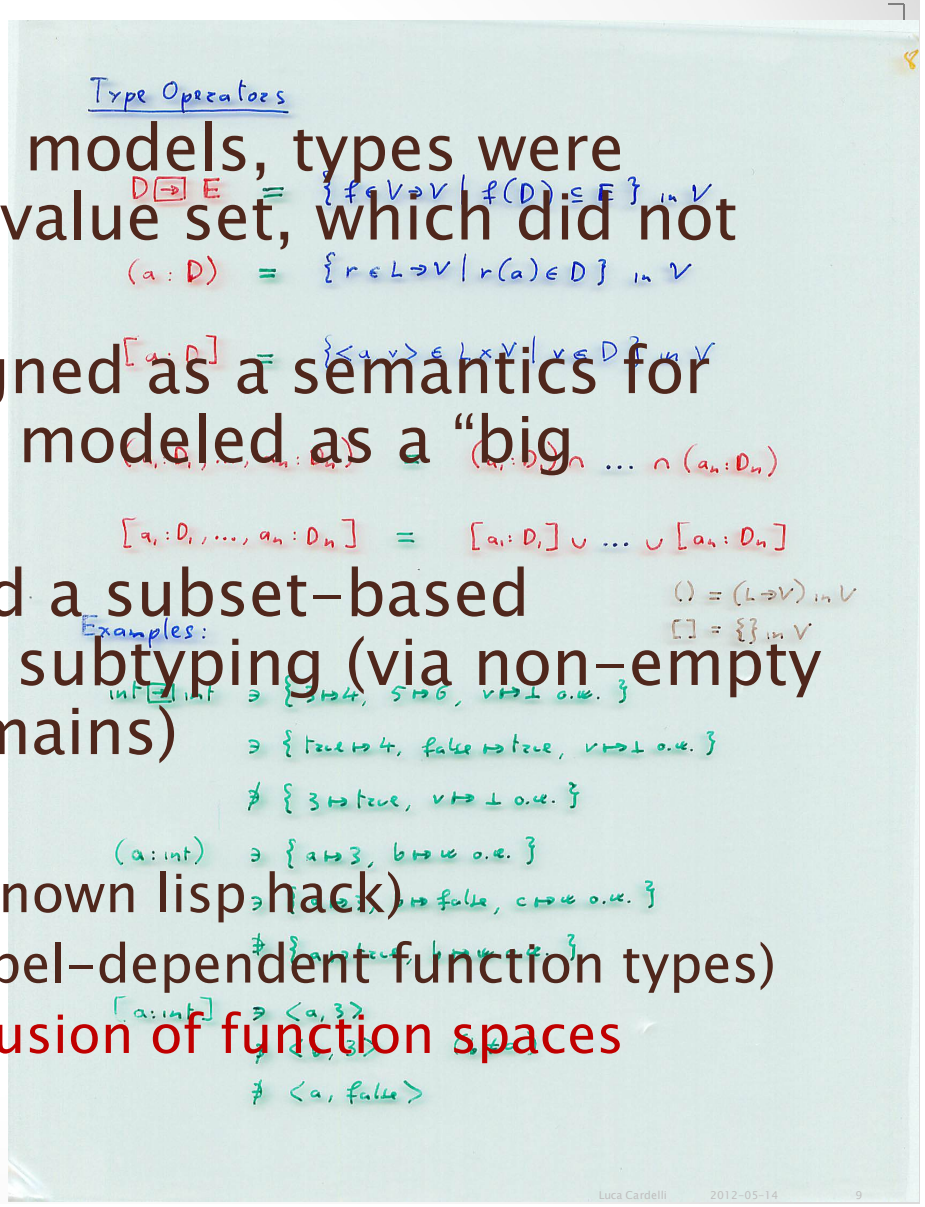
- The Practice of Objects
  - Pisa: Simula was my favorite language.
  - Edinburgh: Added records and variants to ML. Tried to add record subtyping by type inference, but gave up.
  - Pisa again: Galileo, an ML-insipired database programming language with records and variants and Simula-inspired subtyping.



- Where was the Theory of Objects?
  - Logic languages: --> Predicate logic
  - Database languages: --> Relational calculus
  - Functional languages: --> λ-calculus
  - Imperative languages: --> Hoare Logic / Weakest Preconditions
  - Modular languages: --> Algebraic semantics
  - Object-oriented languages: --> ???

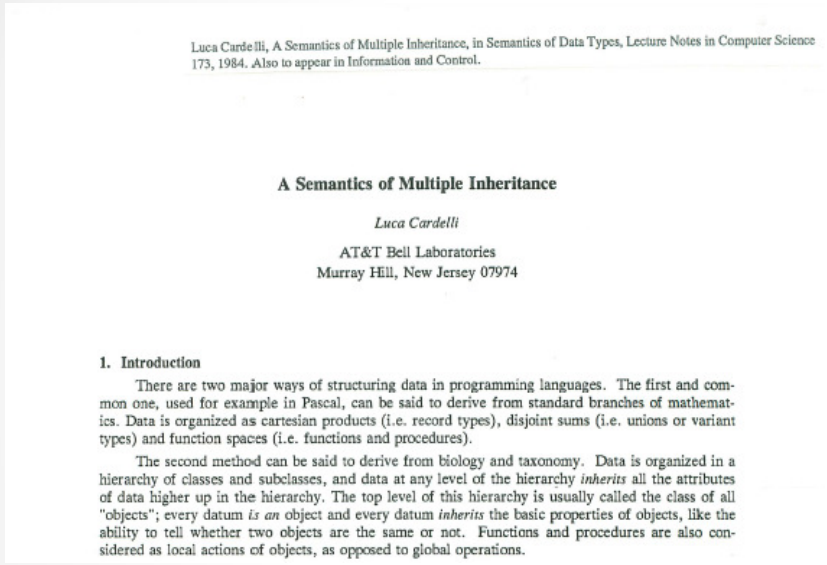
Inspiration!

- The Ideal Model
  - In early Scott denotational models, types were "retracts" of the universal value set, which did not support subtyping.
  - The Ideal Model was designed as a semantics for polymorphism, which was modeled as a "big intersection" of domains.
  - So, it accidentally provided a subset-based denotational semantics of subtyping (via non-empty intersections between domains)
  - Therefore enabling:
    - records as functions (well-known lisp hack)
    - record types as domains (label-dependent function types)
    - record subtyping as set inclusion of function spaces



Result

- Paper:



Luca Cardelli, A Semantics of Multiple Inheritance, in Semantics of Data Types, Lecture Notes in Computer Science 173, 1984. Also to appear in Information and Control.

A Semantics of Multiple Inheritance

Luca Cardelli

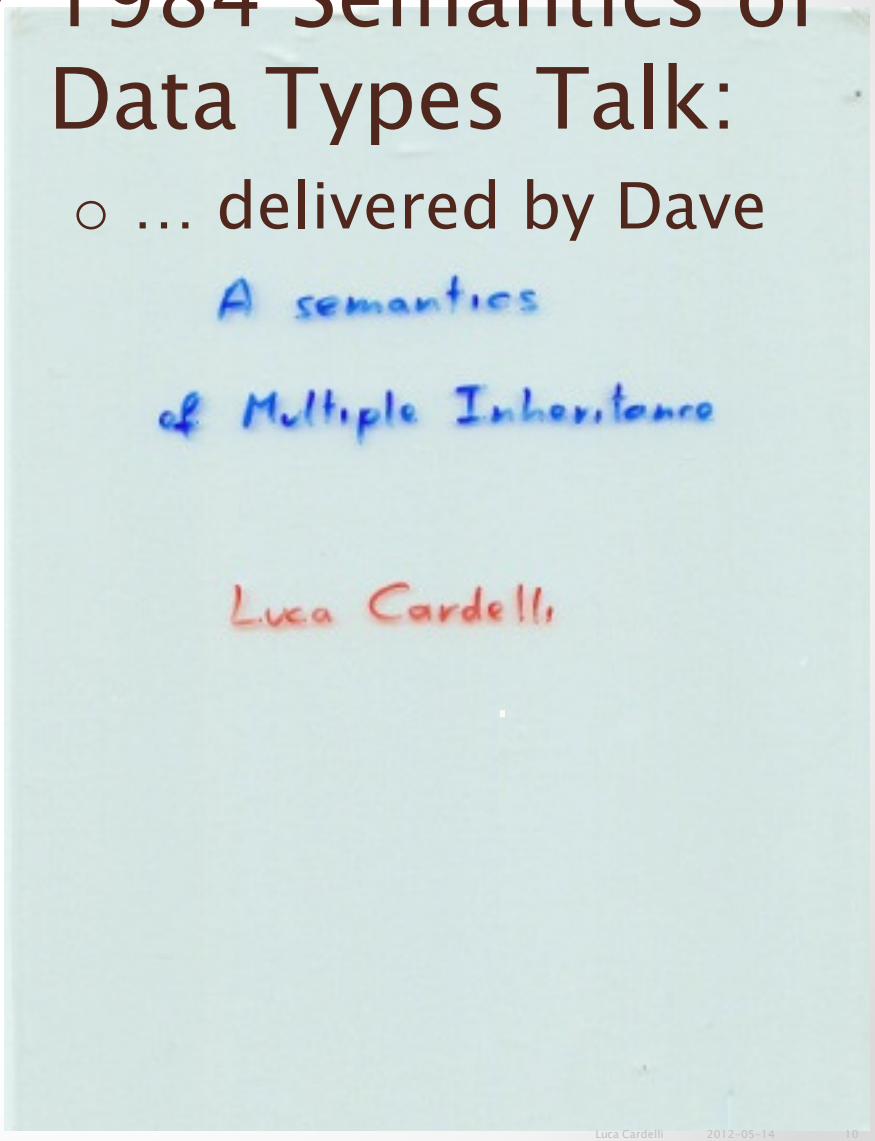
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

There are two major ways of structuring data in programming languages. The first and common one, used for example in Pascal, can be said to derive from standard branches of mathematics. Data is organized as cartesian products (i.e. record types), disjoint sums (i.e. unions or variant types) and function spaces (i.e. functions and procedures).

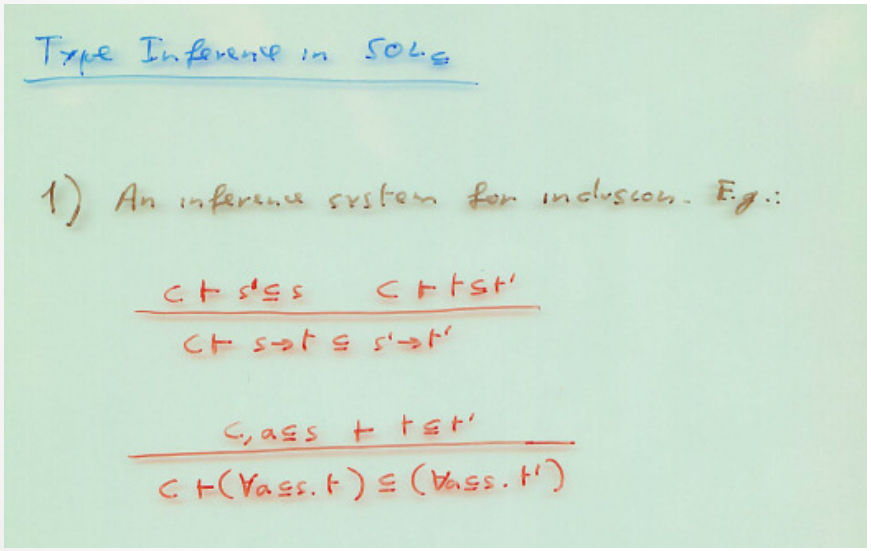
The second method can be said to derive from biology and taxonomy. Data is organized in a hierarchy of classes and subclasses, and data at any level of the hierarchy *inherits* all the attributes of data higher up in the hierarchy. The top level of this hierarchy is usually called the class of all "objects"; every datum *is an* object and every datum *inherits* the basic properties of objects, like the ability to tell whether two objects are the same or not. Functions and procedures are also considered as local actions of objects, as opposed to global operations.

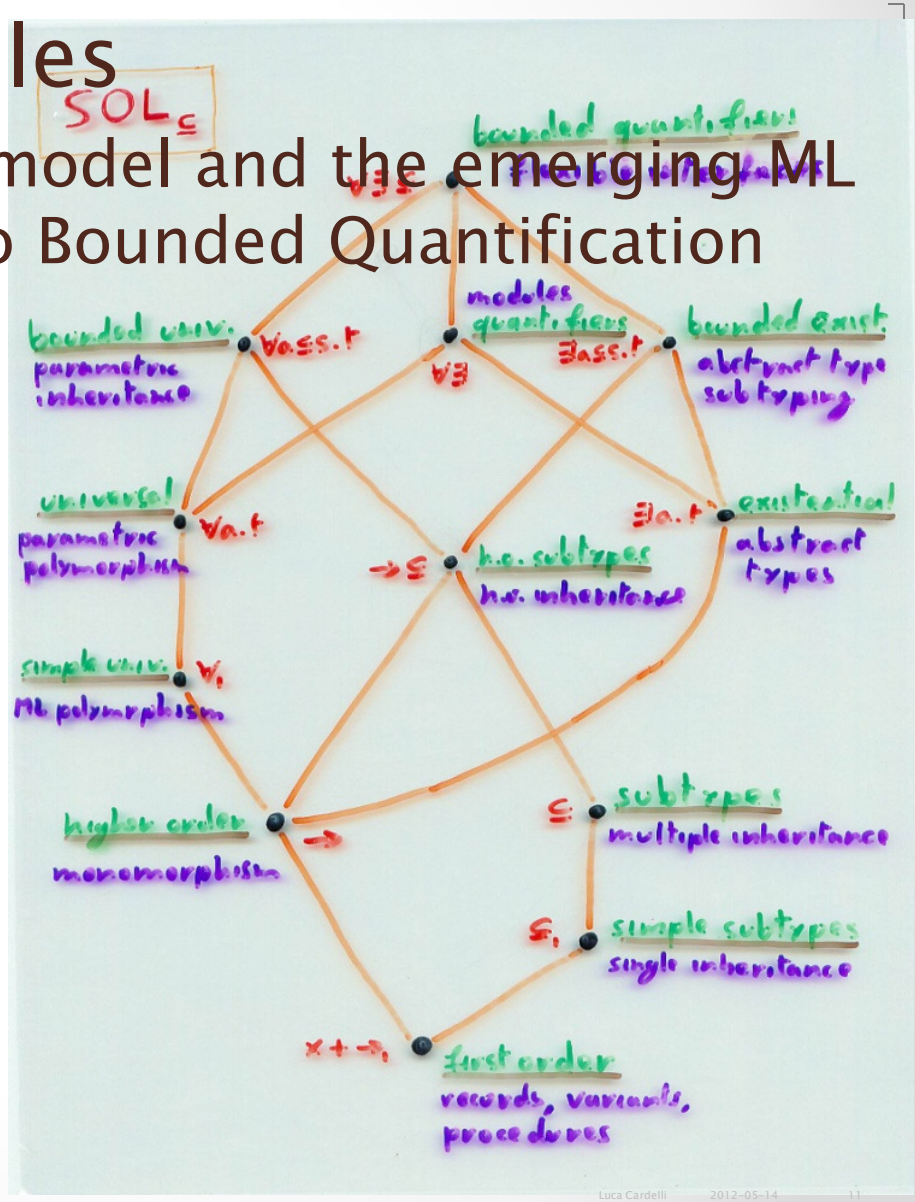
- 1984 Semantics of Data Types Talk:
  - … delivered by Dave



Followup

- Quantifiers and Modules
  - Still inspired by the Ideal model and the emerging ML module system, leading to Bounded Quantification





One Joint Paper

Persistence and Type Abstraction

Luca Cardelli
David MacQueen

AT&T Bell Laboratories
Murray Hill, NJ 07974

Introduction

Abstract types are a well known and effective way of structuring programs. The basic idea of information hiding can however conflict with the need to store data for long periods of time, and make it accessible to different activities. In particular a typechecker must be able to recognize the occurrence of the *same* abstract type during different activations, and must enforce the privacy of data representations.

To achieve this, the persistent storage of data must preserve type information, and must respect type abstraction. The use of type abstractions in the presence of persistent storage requires that abstract types be made persistent as well. Under these conditions, we can preserve type security across distinct activations of the typechecker.

The following is a brief account of how various models of abstraction and persistence interact. We start by sketching a simple polymorphic language and its types and showing various ways of modeling type abstraction in such a language. We then discuss some basic notions underlying persistent storage of typed objects, such as the intern and extern primitives and the special type *dynamic*, and describe three persistence strategies. Finally we discuss the particular problem of persistent abstract types.

Values and Types

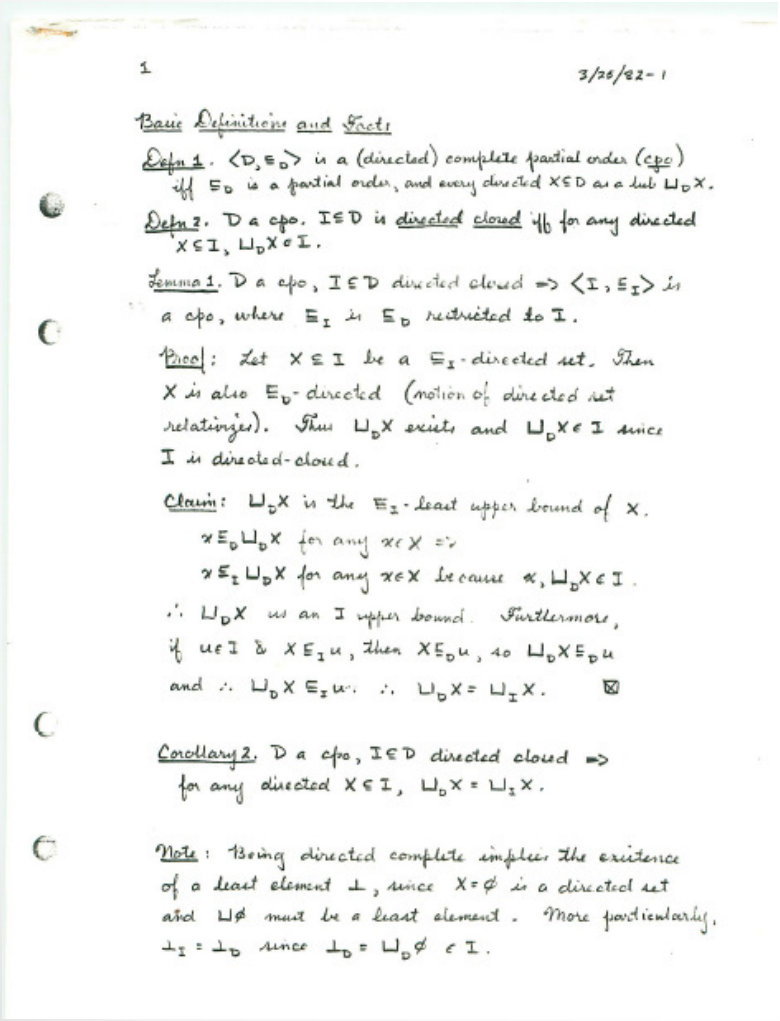
We will base our discussion on a simple polymorphic language in the tradition of ML [Milner 84] and Amber [Cardelli 84]. The simplified language we have in mind is closely related to the language SOL [Mitchell and Plotkin 85], variants of which are described in [Reynolds 85] and [Cardelli and Wegner 85].

The basis of this language is a slightly sugared applied lambda calculus that is adequate for expressing certain kinds of values. Type annotations are added to the basic expressions in such a way that one can statically determine a type for each expression. This type is a structure

[PDF] **Persistence** and type **abstraction**
L Cardelli... - Proceedings of the **Persistence** and ..., 1985 - lucacardelli.name
Page 1. 231 **Persistence** and Type **Abstraction** Luca Cardelli David MacQueen AT&T Bell Laboratories Murray Hill, NJ **...** The basic ideas of information hiding can however conflict with the need to store **data** for long periods of time, and make it accessible to different activities. **...**
Cited by 34 - Related articles - All 8 versions

Archive

1982-03 “Basic Definitions and Facts” (?) 48 pages



1 3/26/82-1

Basic Definitions and Facts

Defn 1. $\langle D, \sqsubseteq_D \rangle$ is a (directed) complete partial order (cpo) iff $\sqsubseteq_D$ is a partial order, and every directed $X \subseteq D$ as a lub $\sqcup_D X$.

Defn 2. D a cpo. $I \subseteq D$ is directed closed if for any directed $X \subseteq I$, $\sqcup_D X \in I$.

Lemma 1. D a cpo, $I \subseteq D$ directed closed $\Rightarrow \langle I, \sqsubseteq_I \rangle$ is a cpo, where $\sqsubseteq_I$ is $\sqsubseteq_D$ restricted to I.

Proof: Let $X \subseteq I$ be a $\sqsubseteq_I$-directed set. Then X is also $\sqsubseteq_D$-directed (notion of directed set relativizes). Thus $\sqcup_D X$ exists and $\sqcup_D X \in I$ since I is directed-closed.

Claim: $\sqcup_D X$ is the $\sqsubseteq_I$-least upper bound of X.

$x \sqsubseteq_D \sqcup_D X$ for any $x \in X$ $\Rightarrow$

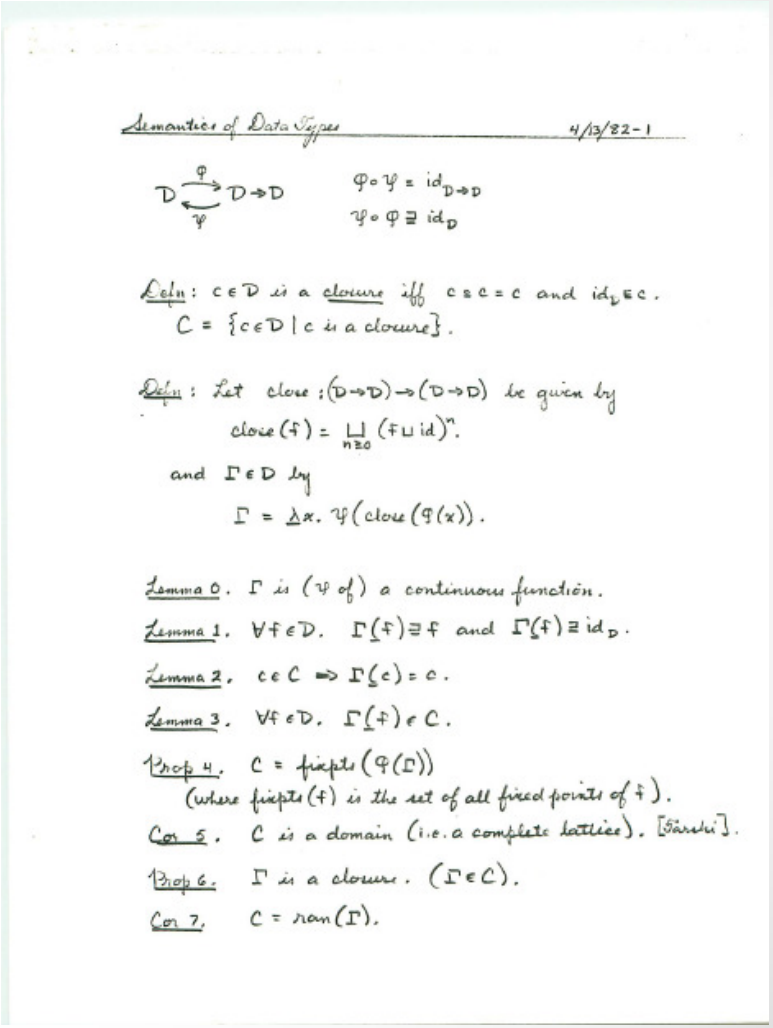
$x \sqsubseteq_I \sqcup_D X$ for any $x \in X$ because $x, \sqcup_D X \in I$.

$\therefore \sqcup_D X$ is an I upper bound. Furthermore, if $u \in I$ & $X \sqsubseteq_I u$, then $X \sqsubseteq_D u$, so $\sqcup_D X \sqsubseteq_D u$ and $\therefore \sqcup_D X \sqsubseteq_I u$. $\therefore \sqcup_D X = \sqcup_I X$. ⊠

Corollary 2. D a cpo, $I \subseteq D$ directed closed $\Rightarrow$ for any directed $X \subseteq I$, $\sqcup_D X = \sqcup_I X$.

Note: Being directed complete implies the existence of a least element $\perp$, since $X = \emptyset$ is a directed set and $\sqcup \emptyset$ must be a least element. More particularly, $\perp_I = \perp_D$ since $\perp_D = \sqcup_D \emptyset \in I$.

1982-04 “Semantics of Data Types”



Semantics of Data Types 4/13/82-1

$D \underset{\psi}{\overset{\varphi}{\rightleftarrows}} D \to D$ $\quad \varphi \circ \psi = id_{D \to D}$, $\psi \circ \varphi \sqsupseteq id_D$

Defn: $c \in D$ is a closure iff $c \sqsubseteq c = c$ and $id_D \sqsubseteq c$.
$C = \{c \in D \mid c \text{ is a closure}\}$.

Defn: Let $close : (D \to D) \to (D \to D)$ be given by
$close(f) = \bigsqcup_{n \geq 0} (f \sqcup id)^n$.
and $\Gamma \in D$ by
$\Gamma = \lambda x.\ \psi(close(\varphi(x)))$.

Lemma 0. Γ is (ψ of) a continuous function.

Lemma 1. $\forall f \in D.\ \Gamma(f) \sqsupseteq f$ and $\Gamma(f) \sqsupseteq id_D$.

Lemma 2. $c \in C \Rightarrow \Gamma(c) = c$.

Lemma 3. $\forall f \in D.\ \Gamma(f) \in C$.

Prop 4. $C = fixpts(\varphi(\Gamma))$
(where $fixpts(f)$ is the set of all fixed points of f).

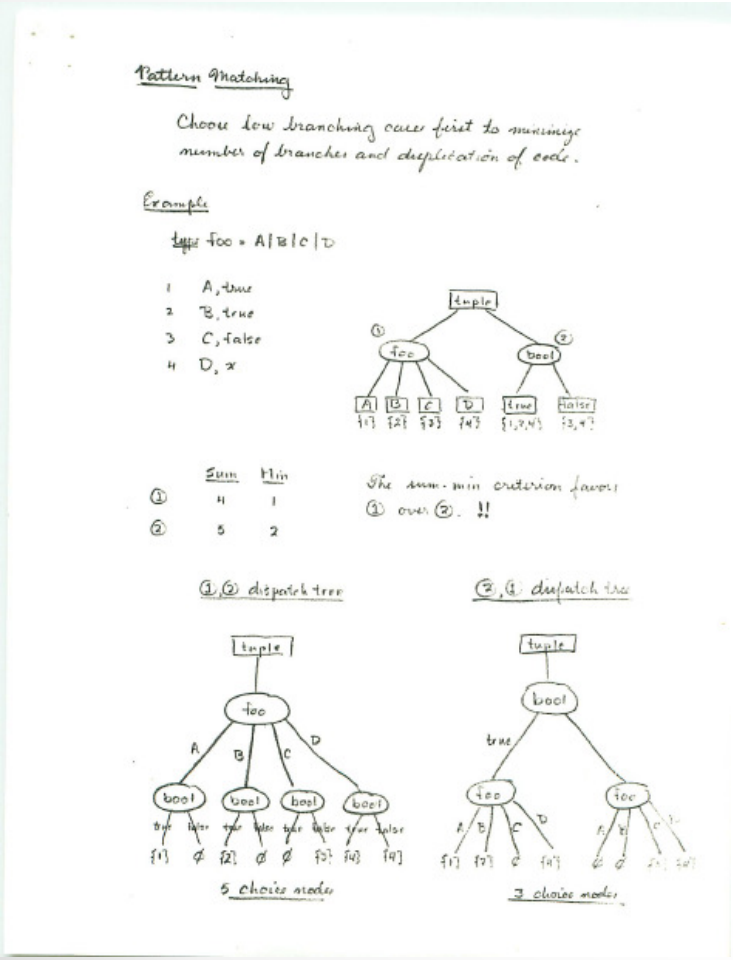
Cor 5. C is a domain (i.e. a complete lattice). [Tarski].

Prop 6. Γ is a closure. ($\Gamma \in C$).

Cor 7. $C = ran(\Gamma)$.

Archive

1983-?? “Pattern Matching”



Pattern Matching

Choose low branching cases first to minimize number of branches and duplication of code.

Example

type foo = A | B | C | D

1 A, true
2 B, true
3 C, false
4 D, ⊥

	Sum	Min
①	4	1
②	5	2

The sum-min criterion favors ① over ②. !!

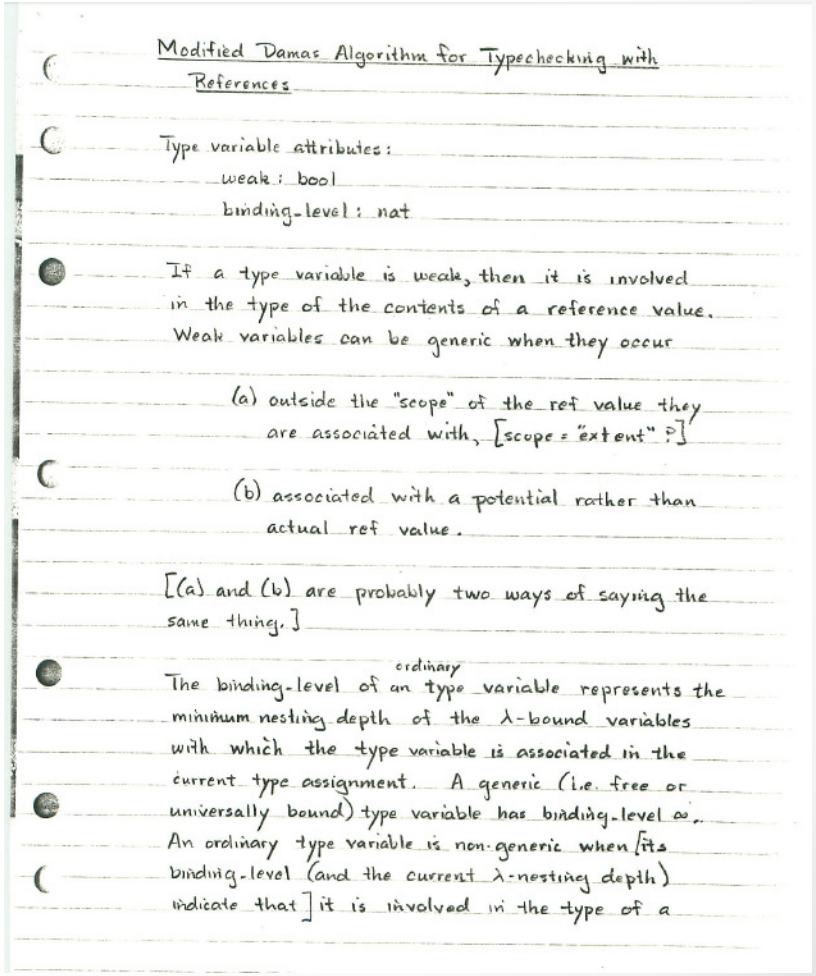
① ② dispatch tree

5 choice nodes

② ① dispatch tree

3 choice nodes

1983-03 “Modified Damas Algorithm for Typechecking with References”



Modified Damas Algorithm for Typechecking with References

Type variable attributes:
weak : bool
binding_level : nat

If a type variable is weak, then it is involved in the type of the contents of a reference value. Weak variables can be generic when they occur

(a) outside the "scope" of the ref value they are associated with, [scope = "extent" ?]

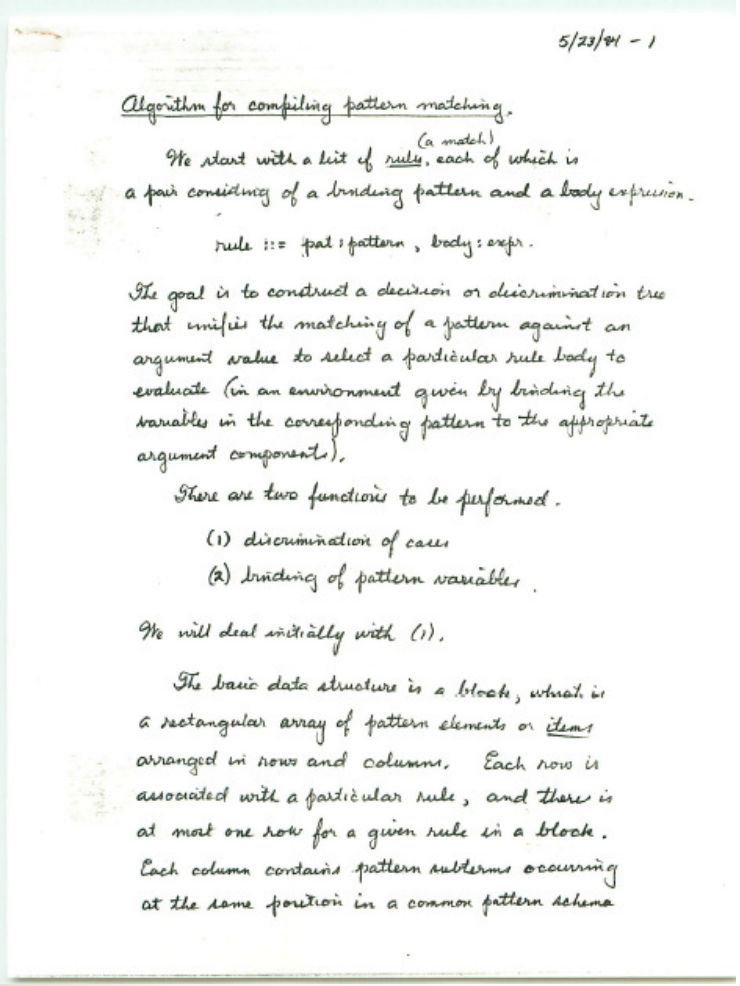
(b) associated with a potential rather than actual ref value.

[(a) and (b) are probably two ways of saying the same thing.]

The binding_level of an ordinary type variable represents the minimum nesting depth of the λ-bound variables with which the type variable is associated in the current type assignment. A generic (i.e. free or universally bound) type variable has binding_level ∞. An ordinary type variable is non-generic when [its binding_level (and the current λ-nesting depth) indicate that] it is involved in the type of a

Archive

1984-05 “Algorithm for Compiling Pattern Matching”



5/23/84 - 1

Algorithm for compiling pattern matching.

We start with a list of rules (a match), each of which is a pair consisting of a binding pattern and a body expression.

rule ::= pat : pattern , body : expr .

The goal is to construct a decision or discrimination tree that unifies the matching of a pattern against an argument value to select a particular rule body to evaluate (in an environment given by binding the variables in the corresponding pattern to the appropriate argument components).

There are two functions to be performed.

(1) discrimination of cases
(2) binding of pattern variables.

We will deal initially with (1).

The basic data structure is a *block*, which is a rectangular array of pattern elements or *items* arranged in rows and columns. Each row is associated with a particular rule, and there is at most one row for a given rule in a block. Each column contains pattern subterms occurring at the same position in a common pattern schema

(The one I failed to implement in my ML compiler.)

End of Story

- → California
  - The Language of Distributed Objects
    - Leading to …
- → England
  - The Language of Mobile Processes
    - Leading to …
  - The Language of Biological Processes
    - Leading to …

The Language of Molecules

Molecular Programming Languages

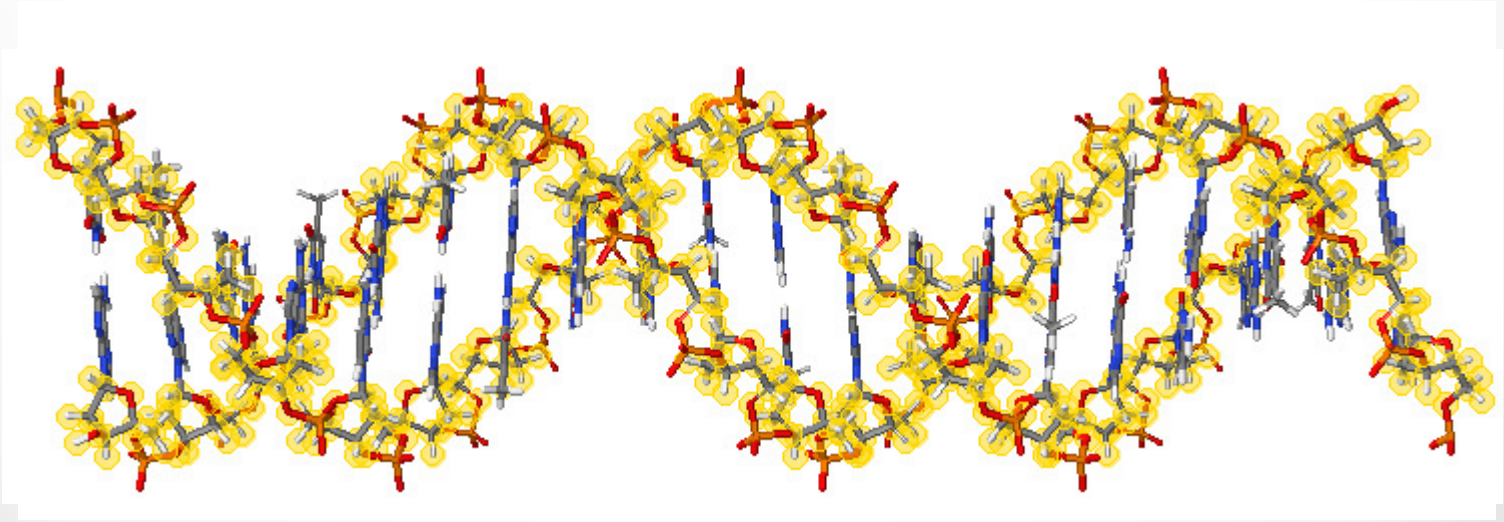
- Reaction-Based (A + B → C + D) (Chemistry)
  - Limited to finite set of species (no polymerization)
  - Practically limited to small number of species (no run-away complexation)
- Interaction-Based (A = !r; C) (Process Algebra)
  - Reduces combinatorial complexity of models by combining independent submodels connected by interactions.
- Rule-Based (A{-}:B{p} → A{p}:B{-}) (Logic, Graph Rewriting)
  - Further reduces model complexity by describing molecular state, and by allowing one to 'ignore the context': a *rule* is a reaction in an unspecified (complexation/phosphorylation) context.
  - Similar to informal descriptions of biochemical events ("narratives").
- Different levels of representation efficiency
  - The latter two can be translated (to each other and) to the first, but doing so may introduce an infinite, or anyway *extremely large*, number of species.

But what about Execution?

- Chemistry is not easily executable
  - Please Mr Chemist, execute me these reactions that I just made up.
- Description
  - Molecular languages used in systems biology are descriptive (modeling) languages
- Compilation
  - How can we compile *arbitrary* molecular programs?
- Execution
  - How can we actually execute molecular languages? With real molecules?

DNA as an Engineering Material

- This is why DNA/RNA is important: it is programmable matter.
- Not the only one, in principle, but the only one for which we have a well-developed manufacturing technology.

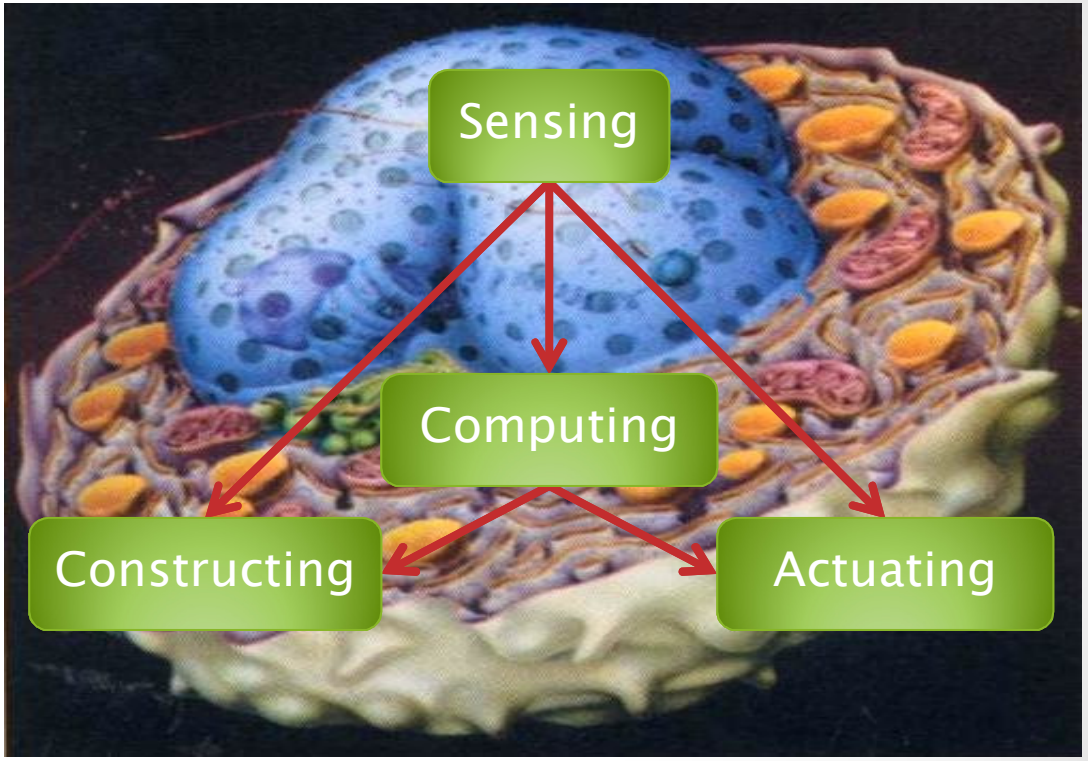


Sequence of Base Pairs (GACT alphabet)

Molecular Control Systems

- Sensing
  - Reacting to forces
  - Binding to molecules
- Actuating
  - Releasing molecules
  - Producing forces
- Constructing
  - Chassis
  - Growth
- Computing
  - Signal Processing
  - Decision Making

Control Systems



Nucleic Acids can do all this.

And interface to biology.

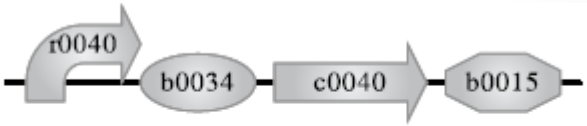
“Embedded” DNA Computing

(Synthetic Biology)

- Using bacterial machinery (e.g.) as the hardware.
  Using embedded gene networks as the software.

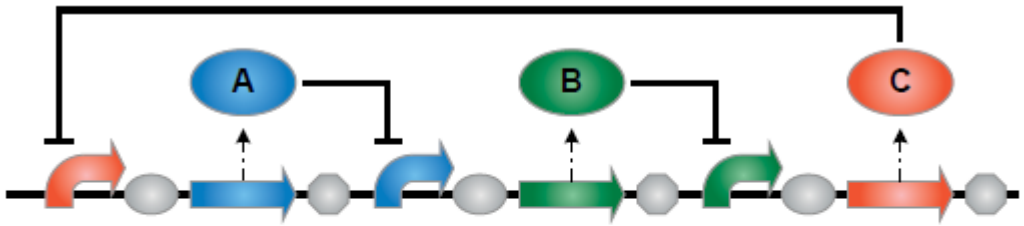
- MIT Registry of Standard Biological Parts

- GenoCAD
  - Meaningful sequences [Cai et al.]




```
r0040:prom; b0034:rbs; c0040:pcr; b0015:ter
```

- GEC
  - [Pedersen & Phillips]



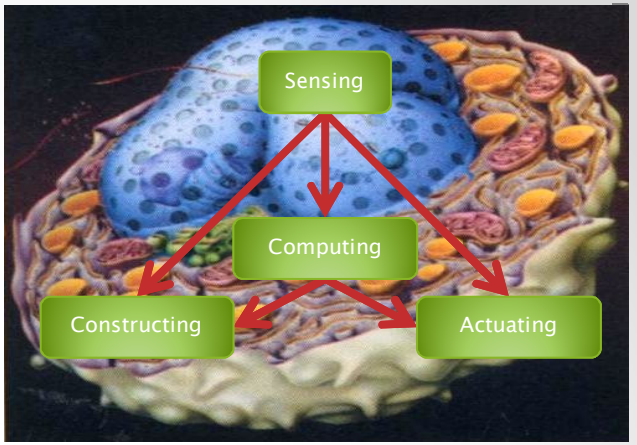

```
prom<neg(C)>; rbs; pcr<codes(A)>; ter;
prom<neg(A)>; rbs; pcr<codes(B)>; ter;
prom<neg(B)>; rbs; pcr<codes(C)>; ter
```

“Autonomous” DNA Computing

(Nano-engineering with biological materials)

- Mix & go
  - All (or most) parts are synthesized
  - No manual cycling (cf. early DNA computing)
  - In some cases, all parts are made of DNA (no enzyme/proteins)
- Self-assembled and self-powered
  - Can run on its own (e.g. environmental sensing)
  - Or be embedded into organisms (in the future)

Curing



A doctor in each cell

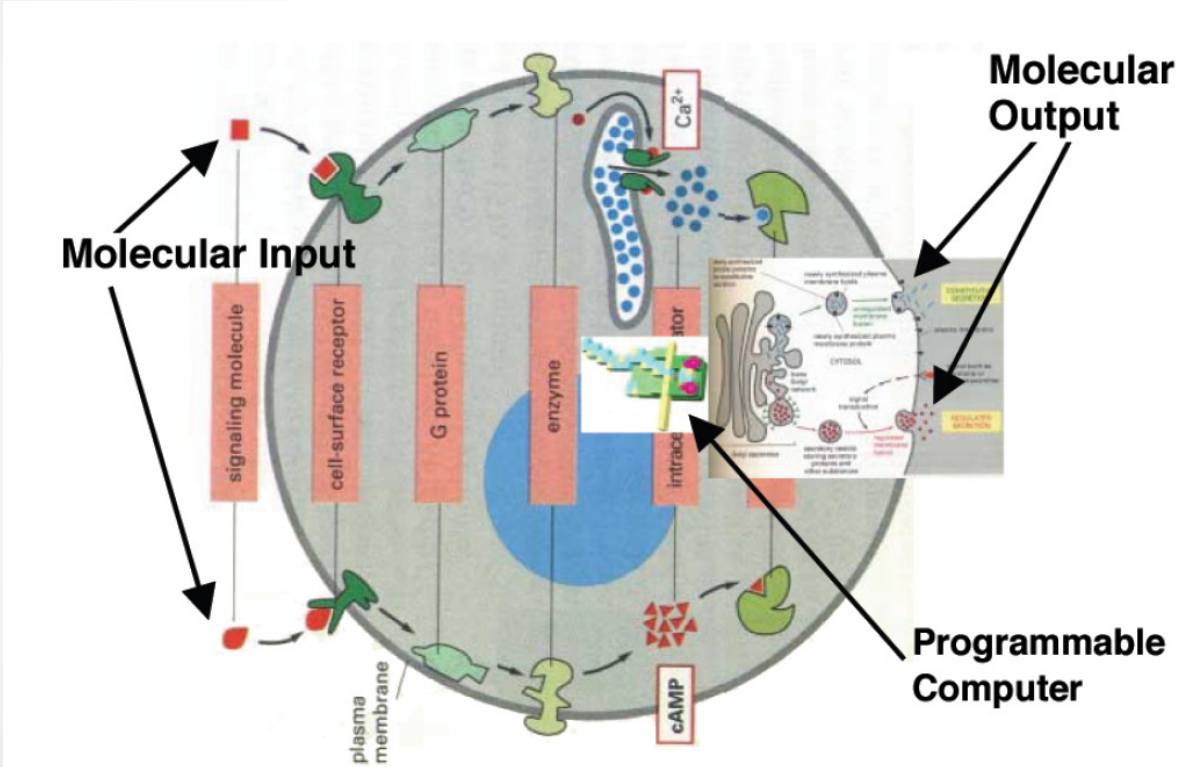


Fig. 1 *Medicine in 2050: "Doctor in a Cell"*

Ehud Shapiro

Rivka Adar
Kobi Benenson
Gregory Linshitz
Aviv Regev
William Silverman

Molecules and computation

Modern DNA Computing

- Non-goals
  - Not to solve NP-complete problems.
  - Not to replace electronic computers.
  - Not necessarily using genes or to producing proteins.

- For general ‘molecular programming’
  - To precisely control the organization and dynamics of matter and information at the molecular level.
  - To interact algorithmically with biological entities.

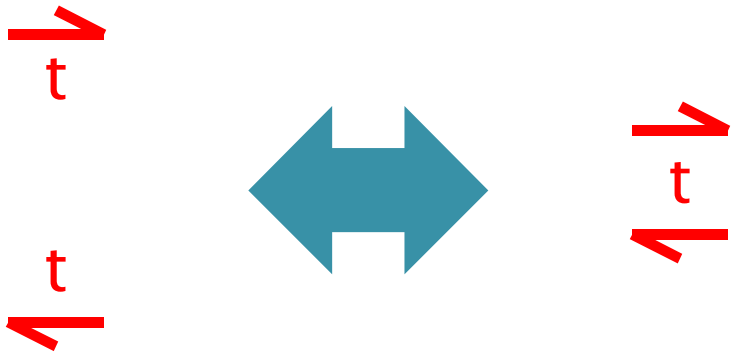
Domains

- Subsequences on a DNA strand are called domains. *PROVIDED* they are "independent" of each other.

- I.e., differently named domains must not hybridize:
  - With each other
  - With each other's complement
  - With subsequences of each other
  - With concatenations of other domains (or their complements)
  - Etc.

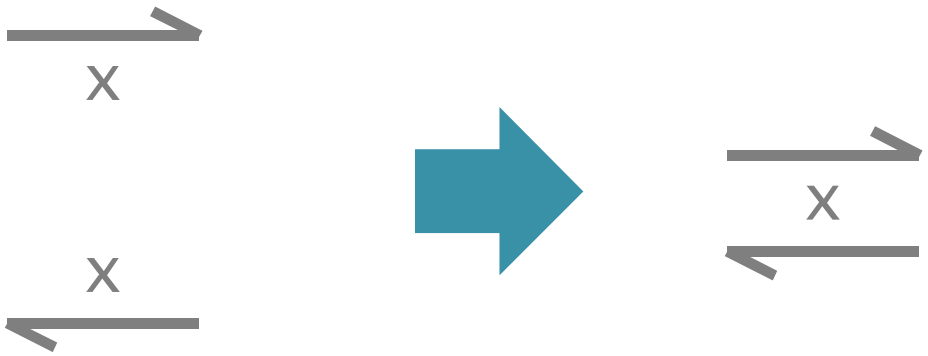
- Choosing domains (subsequences) that are suitably independent is a tricky issue that is still somewhat of an open problem (with a vast literature). But it can work in practice.

Short Domains



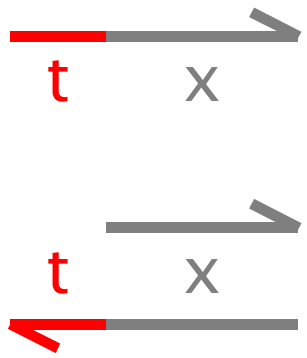
Reversible Hybridization

Long Domains



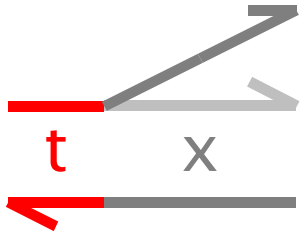
Irreversible Hybridization

Strand Displacement



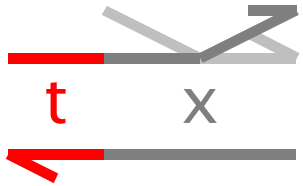
"Toehold Mediated"

Strand Displacement



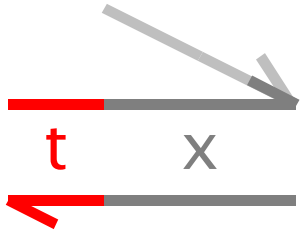
Toehold Binding

Strand Displacement



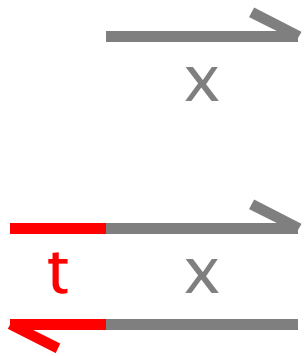
Branch Migration

Strand Displacement



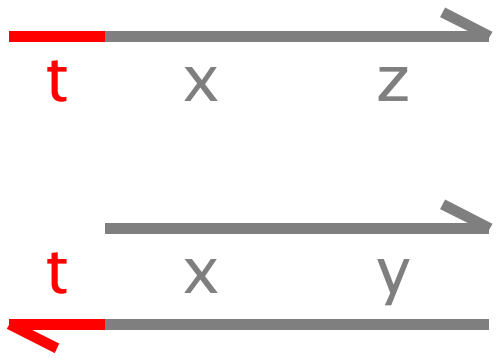
Displacement

Strand Displacement

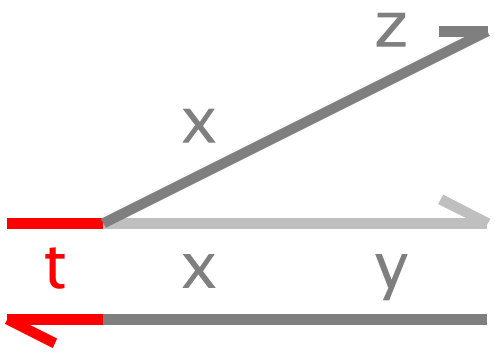


Irreversible release

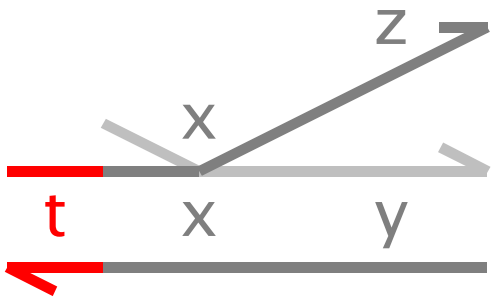
Bad Match



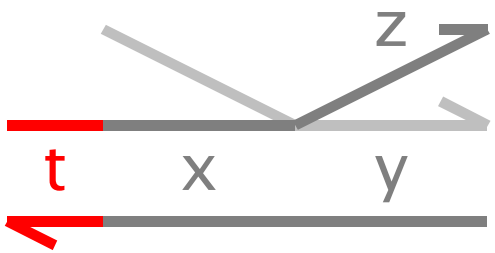
Bad Match



Bad Match



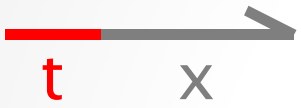
Bad Match



Cannot proceed
Hence will undo

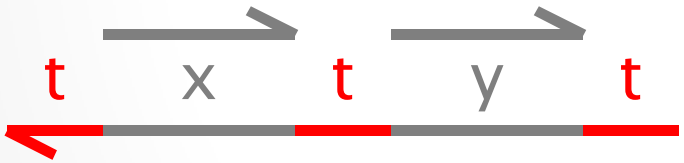
Two-Domain Architecture

- Signals: 1 toehold + 1 recognition region



Garbage collection
"built into" the gates

- Gates: "top-nicked double strands"
(or equivalently double strands with open toeholds)

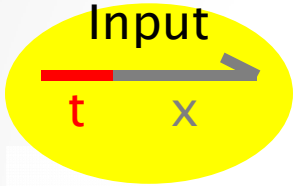


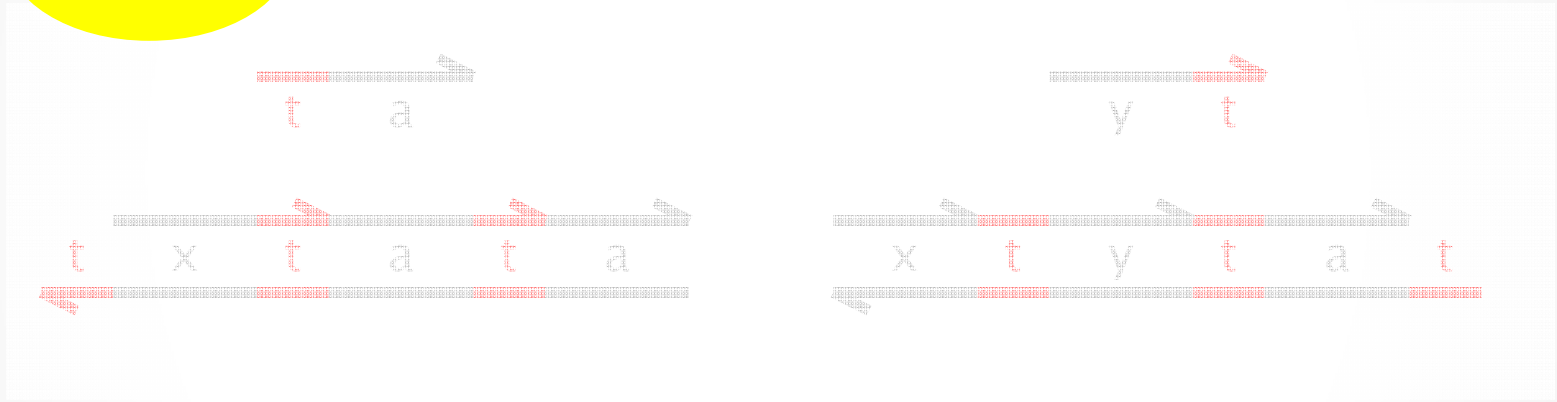
Two-Domain DNA Strand Displacement

Luca Cardelli

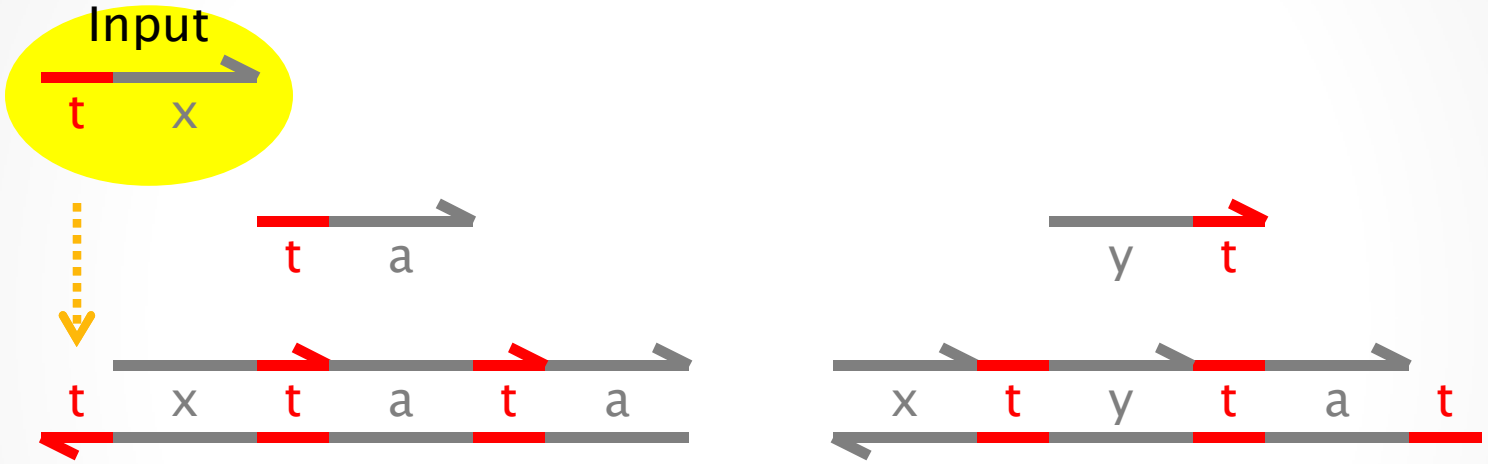
In S. B. Cooper, E. Kashefi, P. Panangaden (Eds.):
Developments in Computational Models (DCM 2010).
EPTCS 25, 2010, pp. 33-47. May 2010.

Transducer x→y





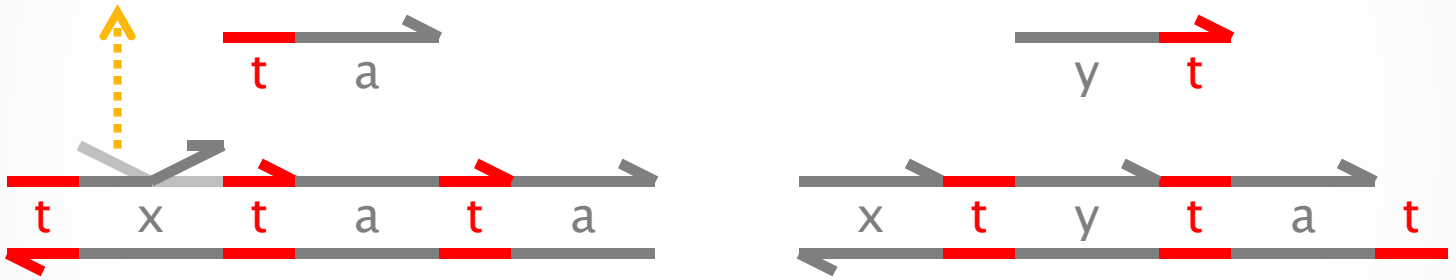
Transducer x→y



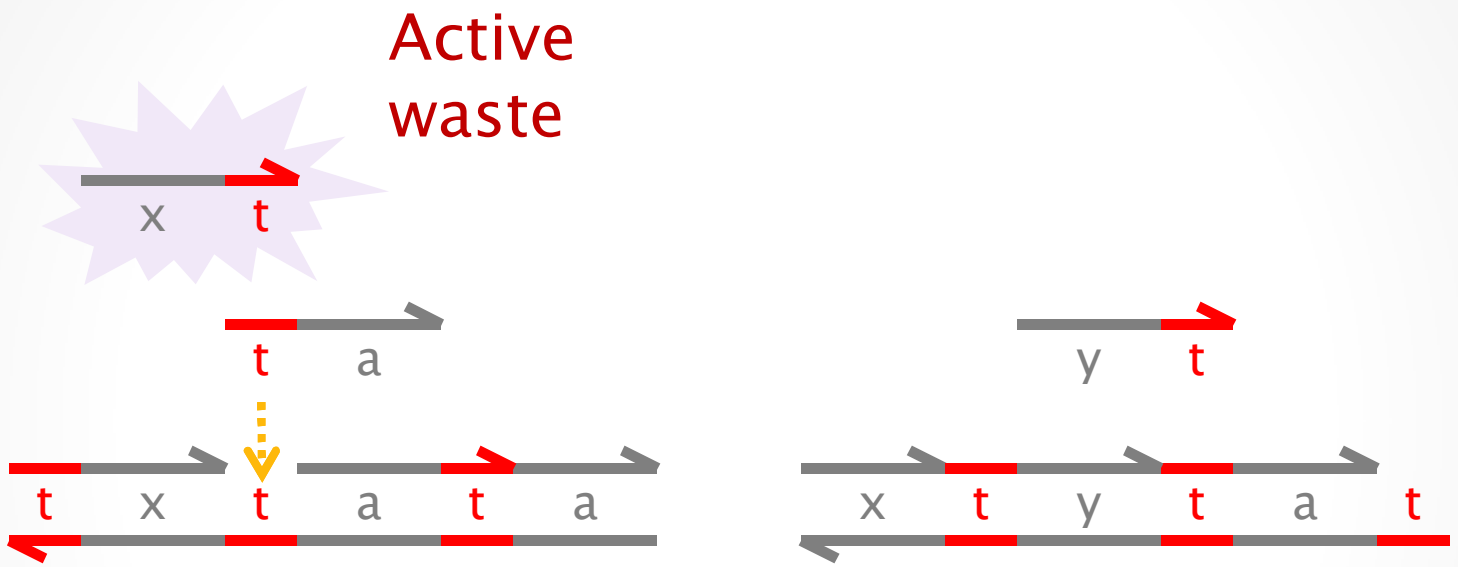
Built by self-assembly!

ta is a *private* signal (a different 'a' for each xy pair)

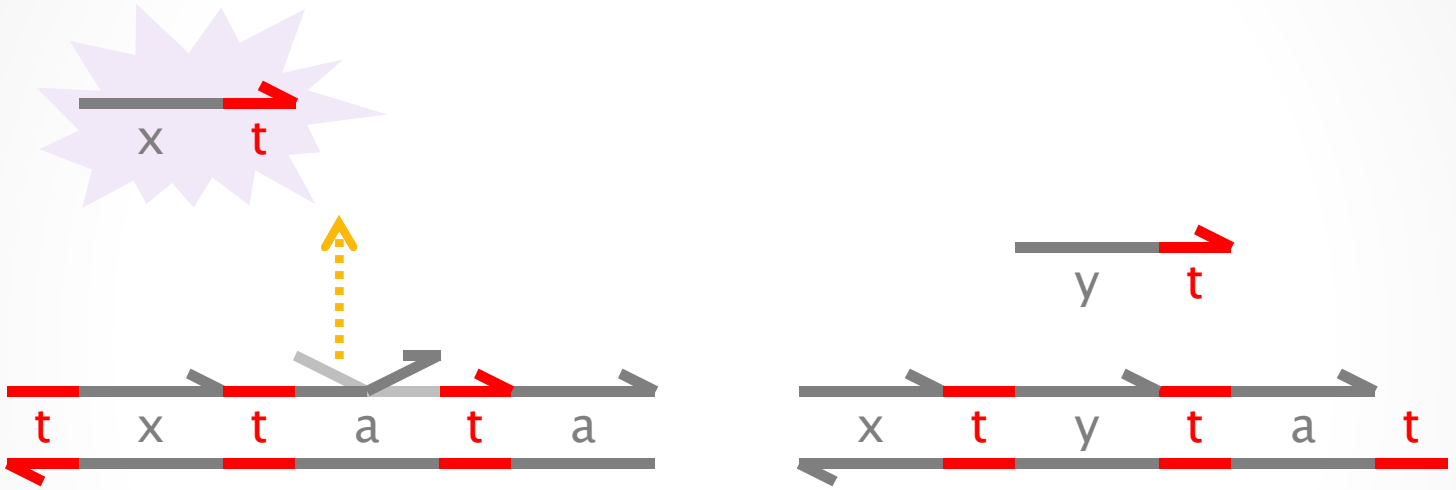
Transducer x→y



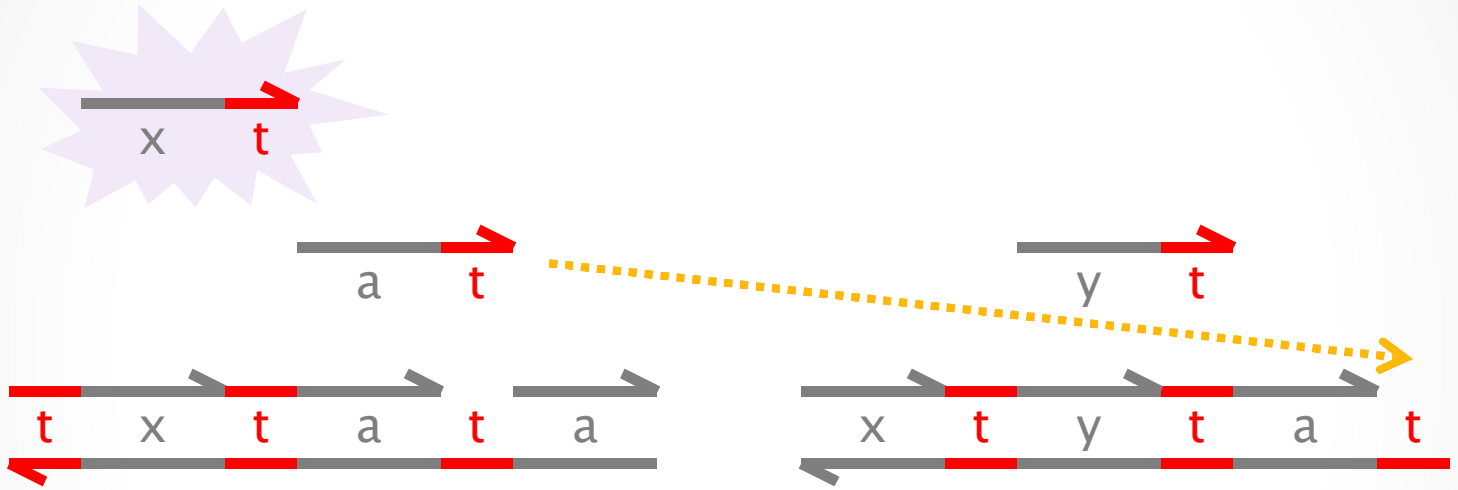
Transducer x→y



Transducer x→y

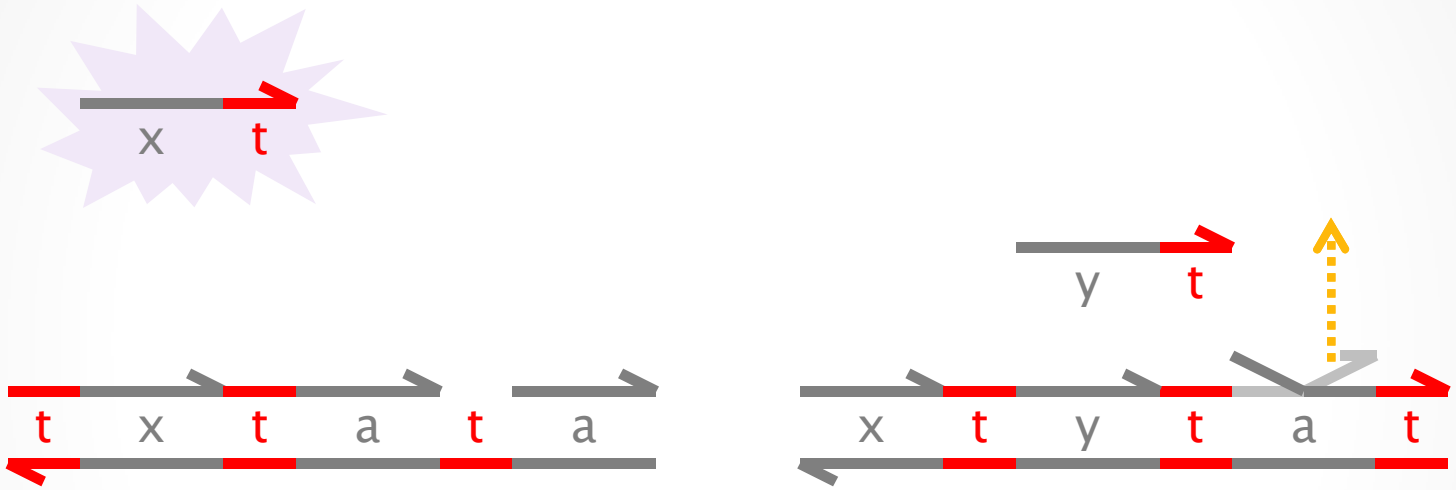


Transducer x→y

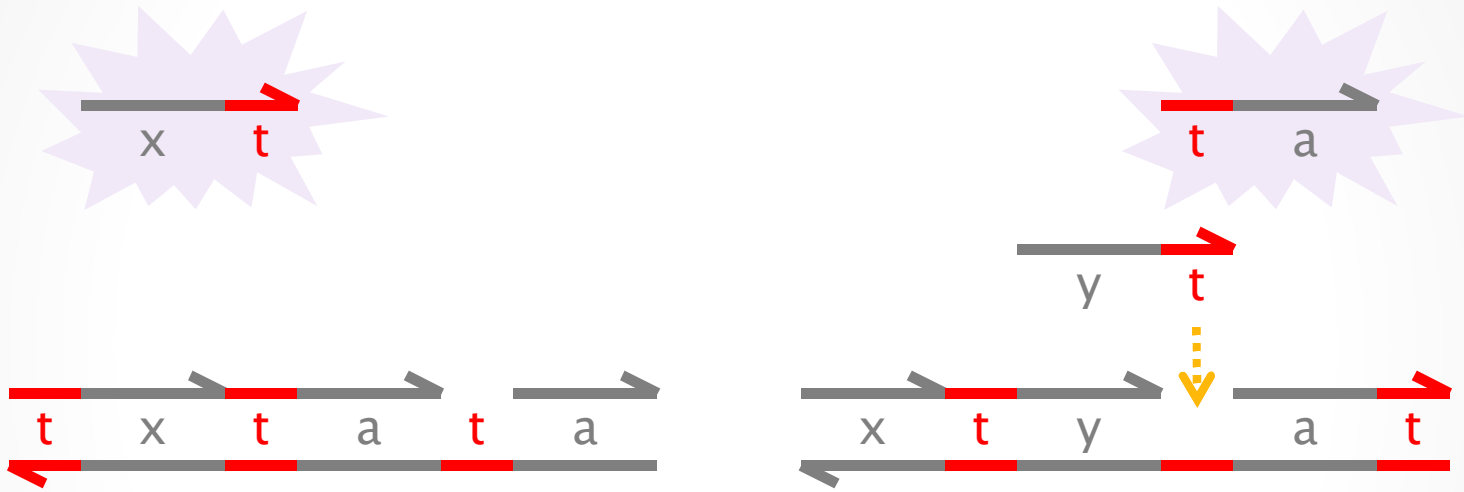


So far, a tx *signal* has produced an at *cosignal.*
But we want signals as output, not cosignals.

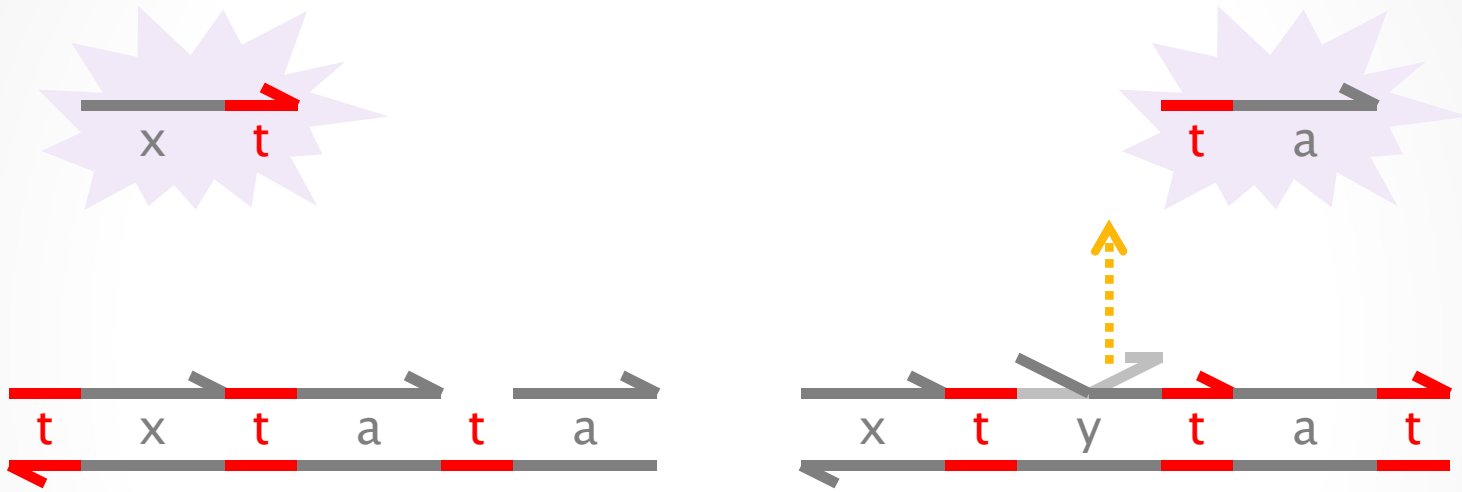
Transducer x→y



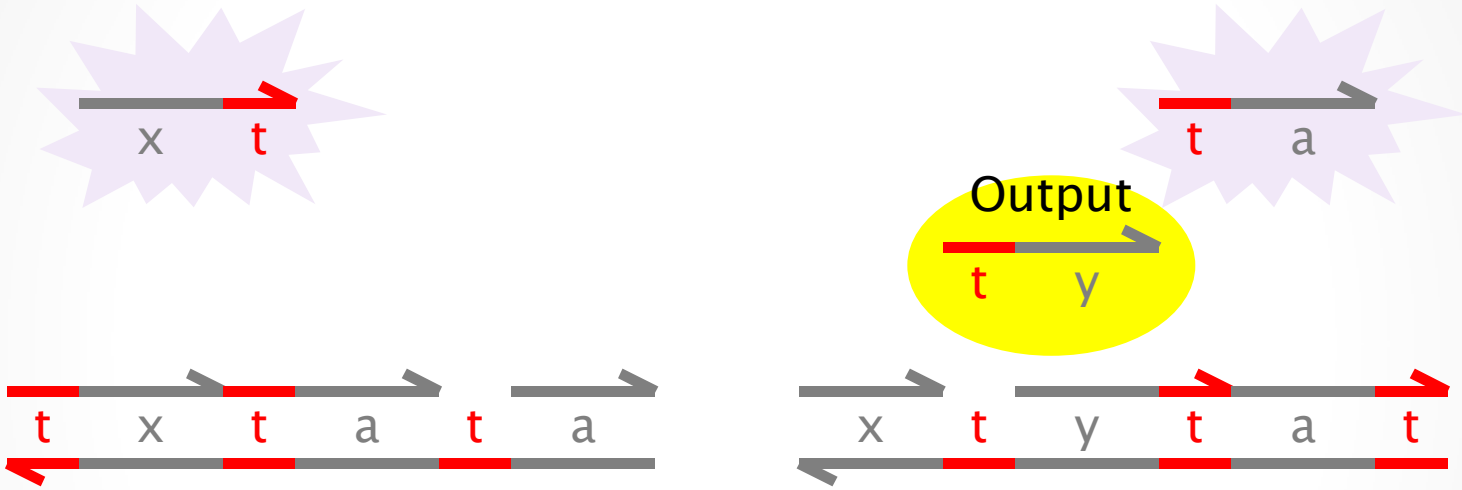
Transducer x→y



Transducer x→y



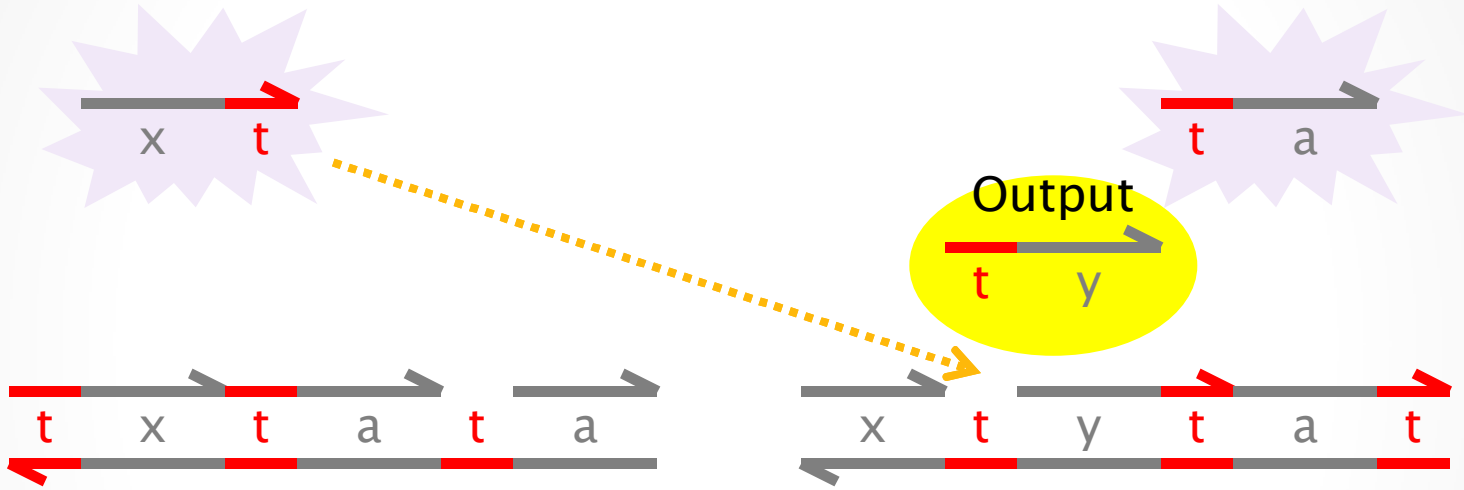
Transducer x→y



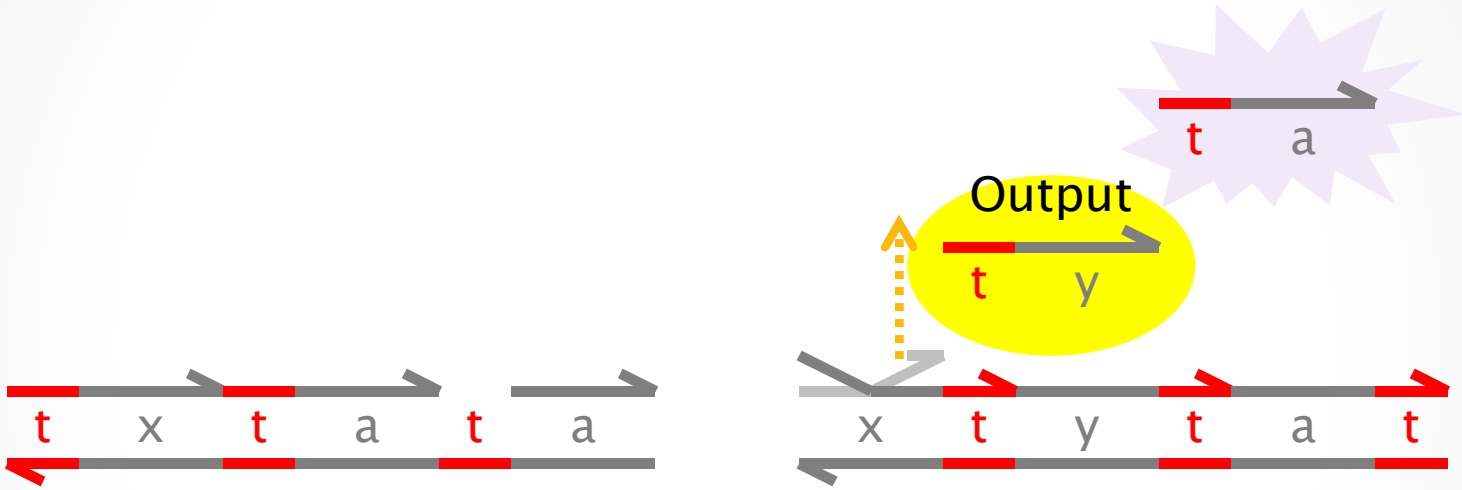
Here is our output ty *signal*.

But we are not done yet:
1) We need to make the output irreversible.
2) We need to remove the garbage.
We can use (2) to achieve (1).

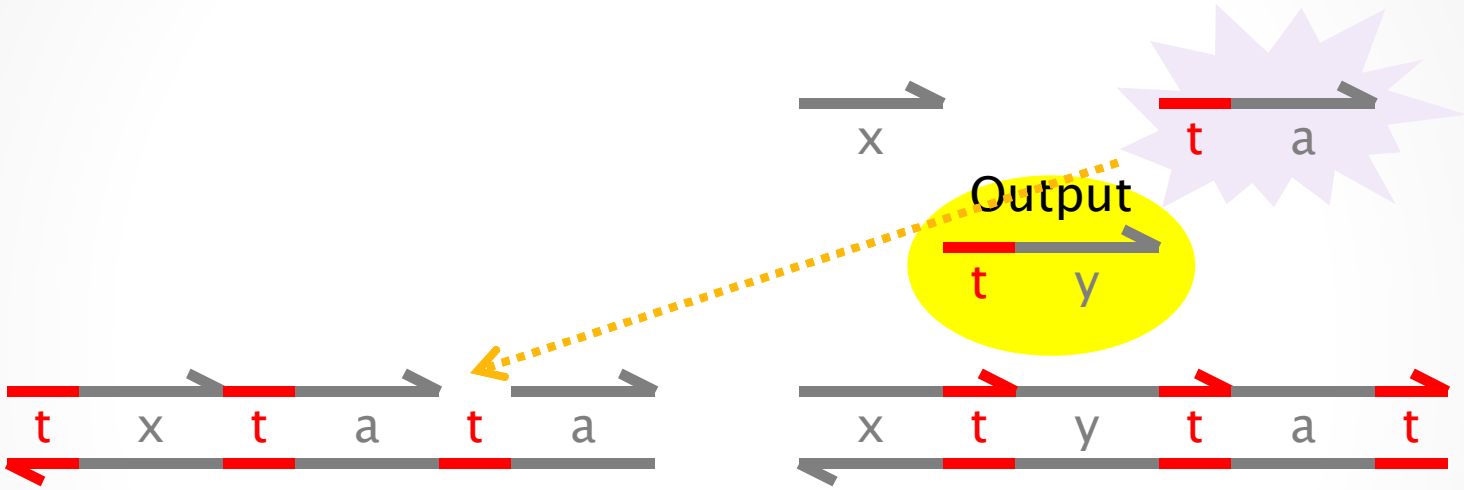
Transducer x→y



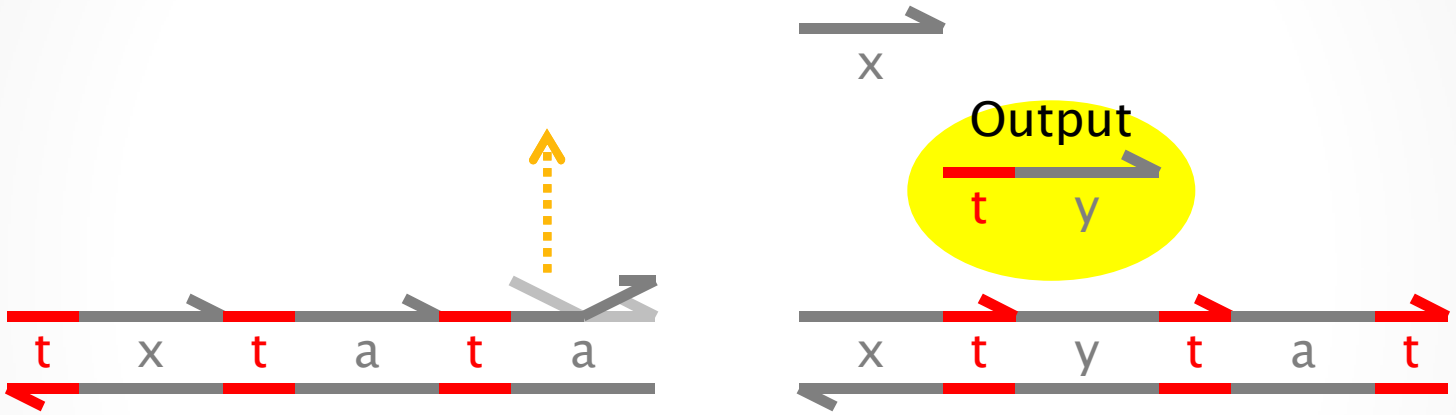
Transducer x→y



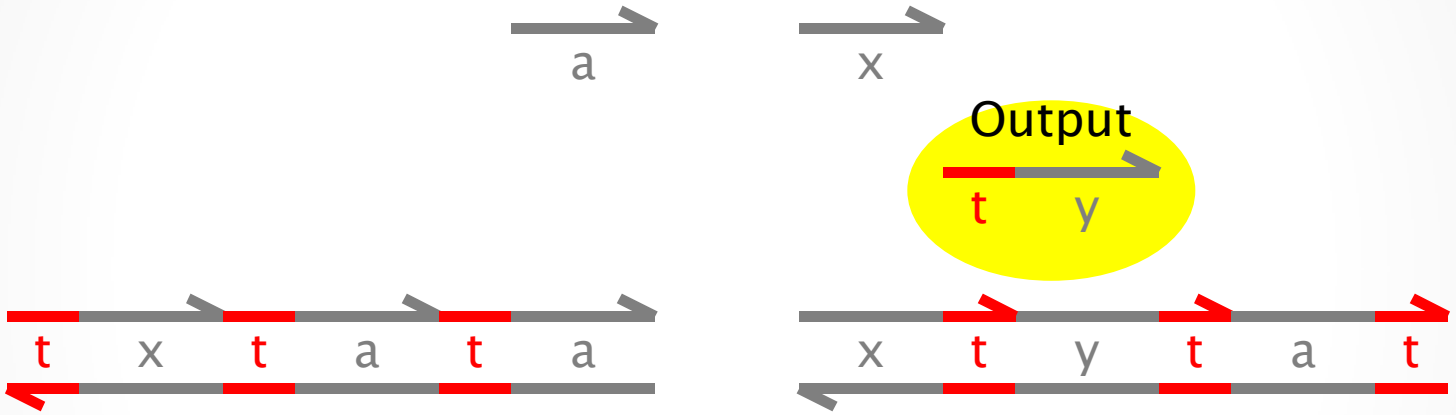
Transducer x→y



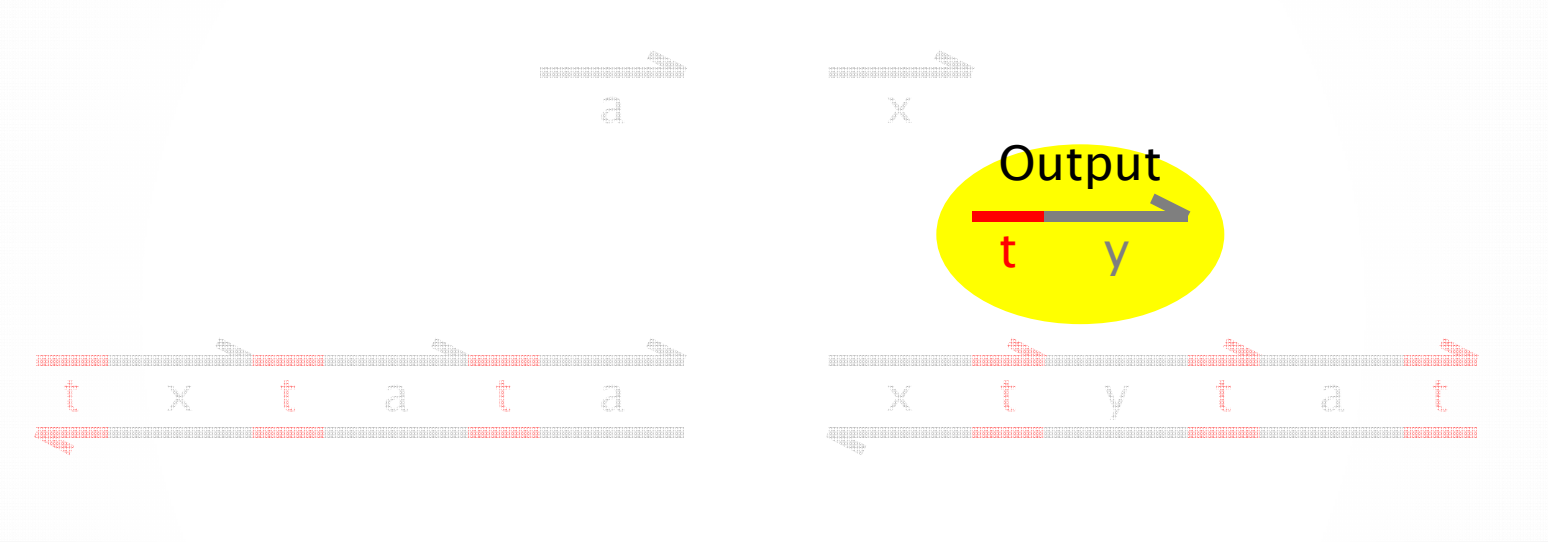
Transducer x→y



Transducer x→y

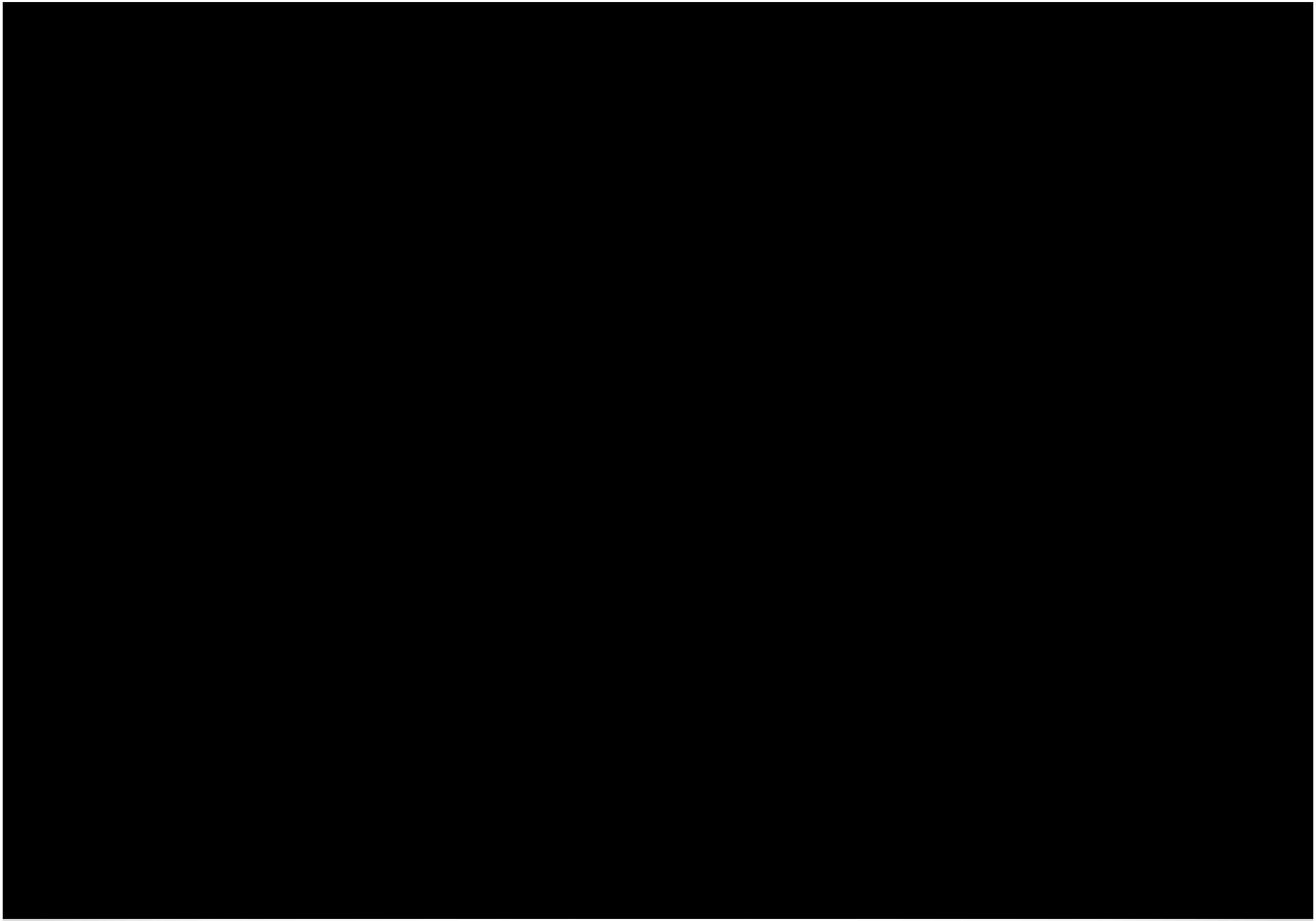


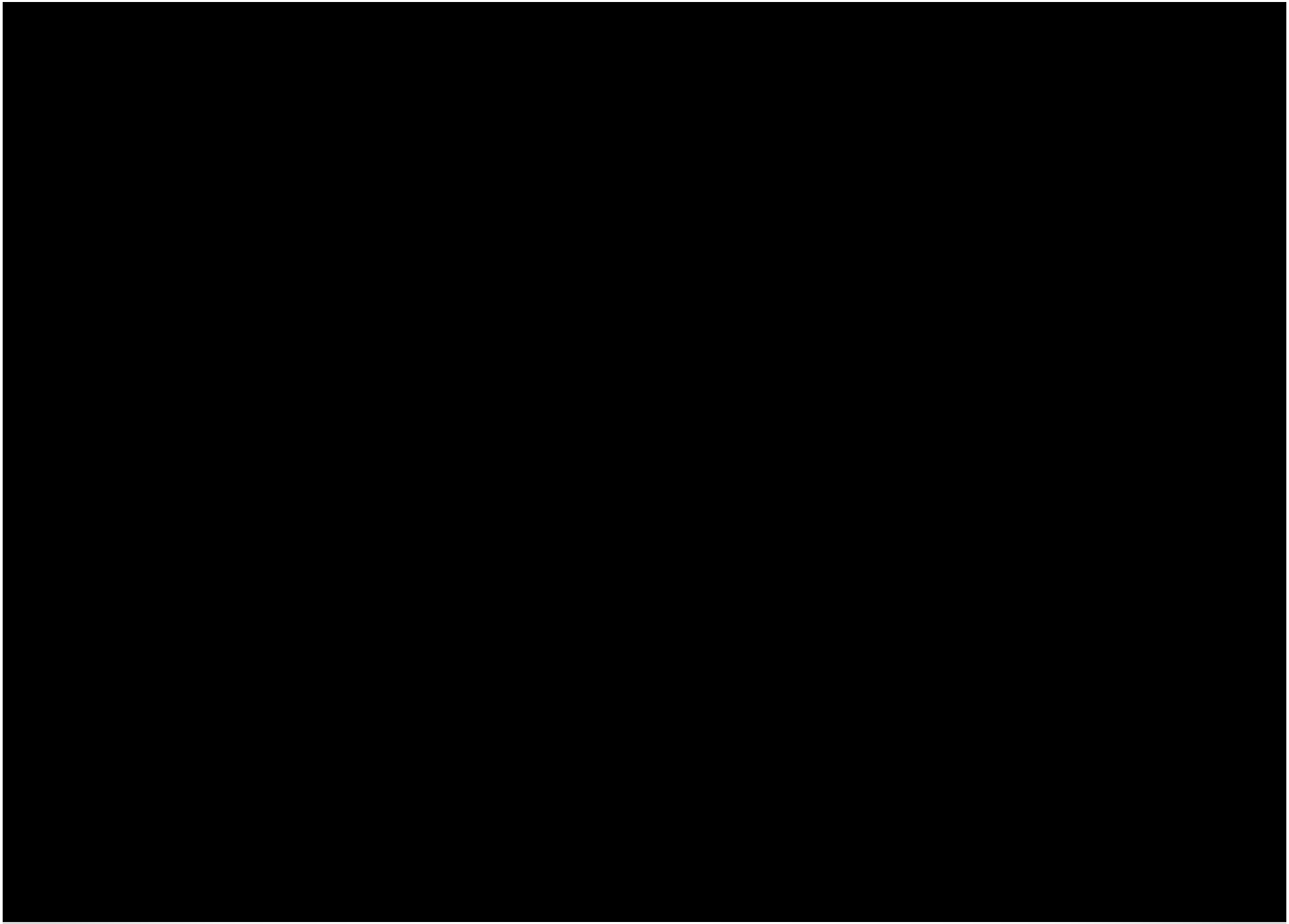
Transducer x→y



Done.

N.B. the gate is consumed: it is the energy source.





General n×m Join-Fork

- Easily generalized to 2+ inputs (with 1+ collectors).
- Easily generalized to 2+ outputs.

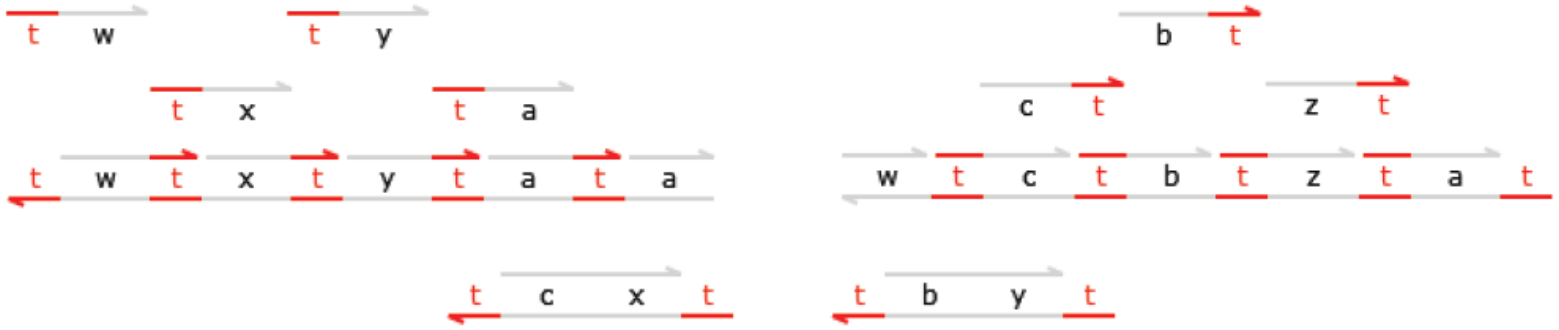
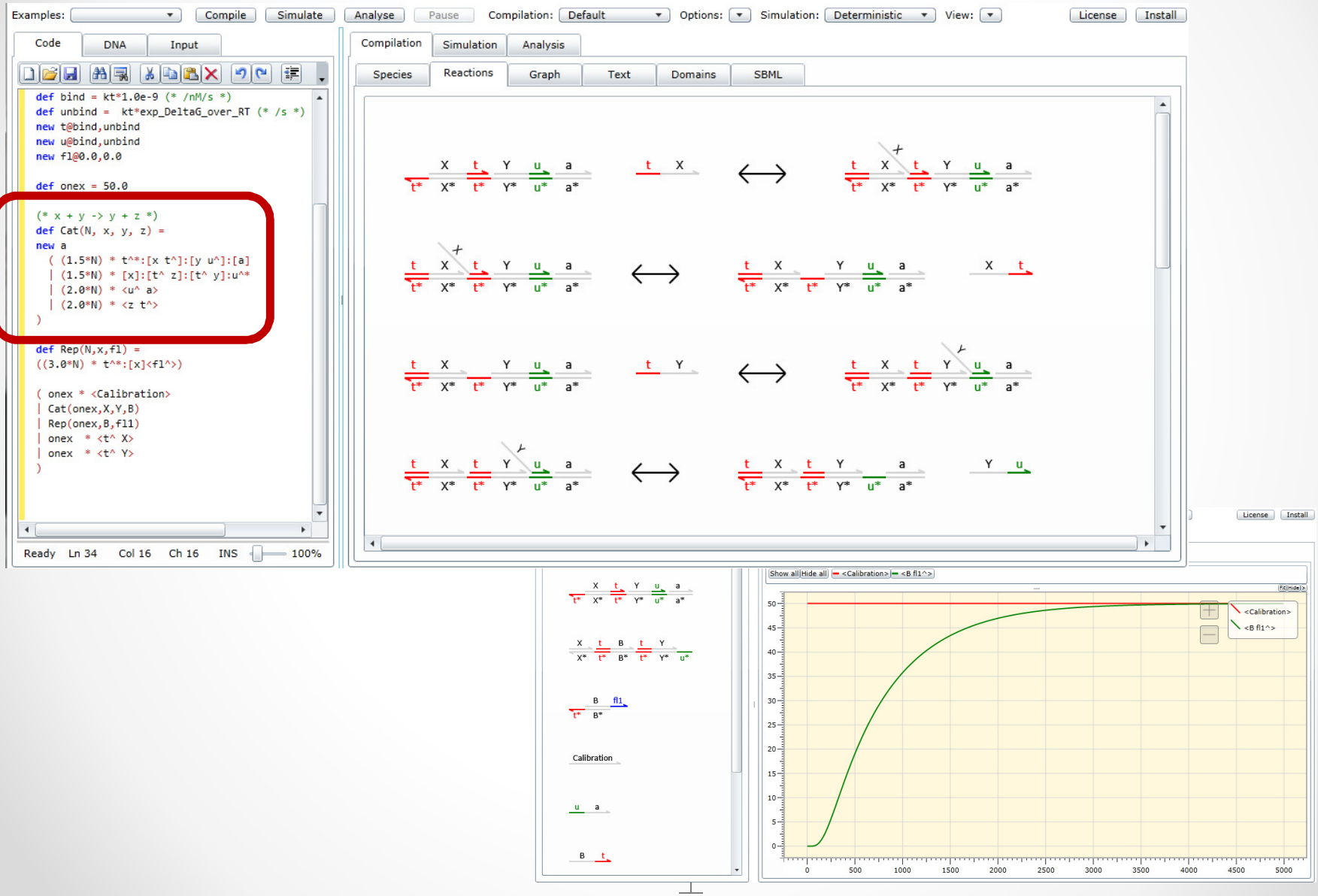


Figure 9: 3-Join $J_{wxyz} \mid tw \mid tx \mid ty \longrightarrow tz$: initial state plus inputs tw, tx, ty.

DNA Programming



Debugging

- Big Networks
  - Two-domain DNA gates for 1 Approximate Majority switch.
  - Initial species: 17
  - Total number of species: 85 (including run-time produced ones)
  - Total number of reactions: 104
- Analysis
  - Gate correctness
  - Circuit correctness
  - Compiler correctness
  - Currently, by simulation
  - Increasingly, by modelchecking:

Design and Analysis of DNA Strand Displacement Devices using Probabilistic Model Checking

Matthew R. Lakin *† David Parker ‡†

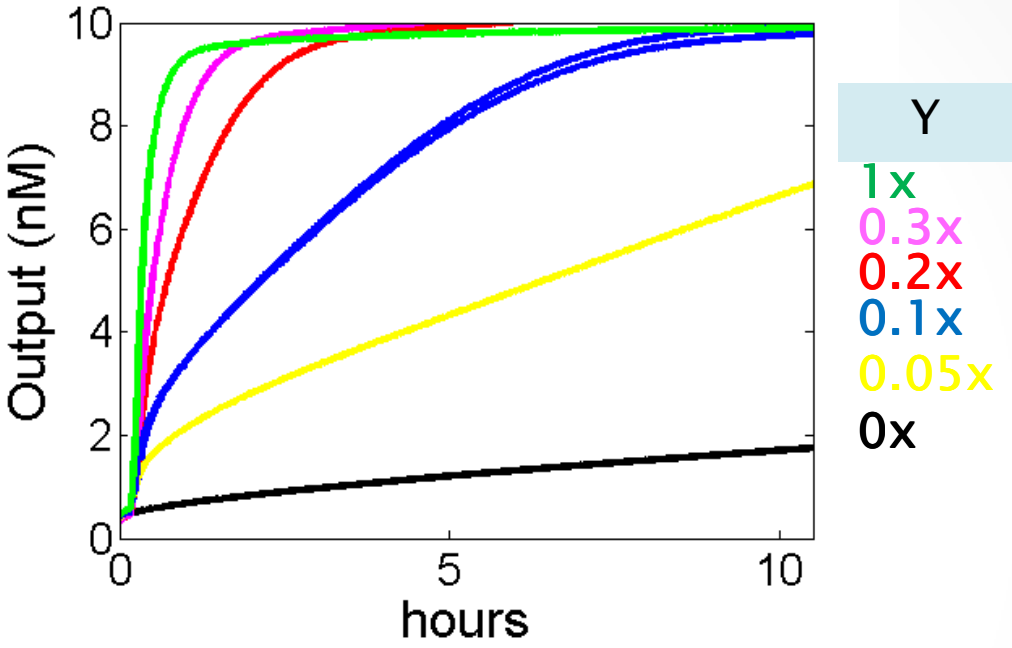
Luca Cardelli* Marta Kwiatkowska ‡

Andrew Phillips*§

Experiments

Two-domain gate for $X+Y \rightarrow Y+B$

X+Y→Y+B
35C
1x = 50nM



Yuan-Jyue Chen and Georg Seelig
U.Washingon.

X+Y→Y+B		Concentration
LG1		1.5x
LG2		1.5x
input		1x
Catalyst		0x, 0.05x,0.1x,0.2x,0.3x,1x
~B		2x
R1		2x
B readout		3x

Summary

- Executable chemistry
  - Given an arbitrary finite chemical network, compile it systematically and execute it.
    [D. Soloveichik, G. Seelig, E. Winfree. DNA as a Universal Substrate for Chemical Kinetics. PNAS 107 no. 12, 5393–5398, 2010.]
  - Finite chemical networks have the computing power of (stochastic) Petri Nets. Population protocols (such as AM) are also well-characterized. [D.Angluin, J.Aspnes, D.Eisenstat, E.Ruppert: The Computational Power of Population Protocols].
- Executable bio-chemistry
  - In addition, DNA supports polymerization, which gives the computing power of Turing Machines.
  - Then the programming language cannot be just chemical reactions, but has to be something more like process algebra or term-rewriting systems.

Conclusions

Conclusions

- The Language of Functions
- …
- …
- The Language of Life