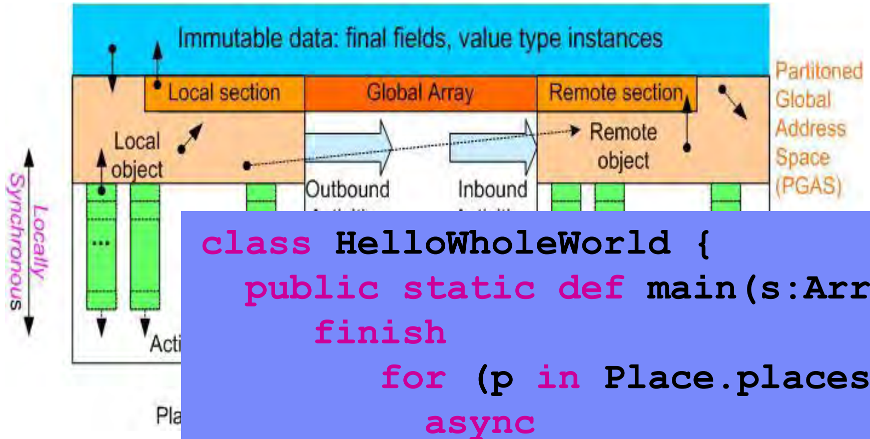
The return of logic

Vijay Saraswat
IBM TJ Watson
July 27, 2011

X10 2.2: An APGAS language



```
class HelloWholeWorld {
  public static def main(s:Array[String]) {
    finish
      for (p in Place.places())
        async
          at (p)
            Console.OUT.println("(At " + p + ") "
                                  + s(0));
  }
}
```

Asychrony
- async S

- atomic S
- when (c) S

Order
- finish S
- clocks

Locality
- at (P) S

- points, regions, distributions, arrays

- Class-based single-inheritance OO
- Structs
- Closures
- True Generic types (no erasures)
- Constrained Types (OOPSLA 08)

established
ws best known
TS upto 3K

- Global Matrix Library shows substantial speedup over Hadoop for data analytics kernels.
- Similar performance improvement for Main Memory Map Reduce engine (M3R) over Hadoop.

Java-like productivity, MPI-like performance

But – how do we handle a billion threads?

- **X10 is (deliberately) low-level**
  - Imperative – explicit mutation, hence very “PC centric” view of computation.
  - Explicit distribution
- **How do you debug a 100,000 threads from a PC-centric point of view?**

- **Our belief**
  - Need to raise level of abstraction
  - Programming model needs to be closer to application domain
  - Implicitly concurrent
  - Statically type safe
  - Declarative
    - Support semantically-based tools, using symbolic reasoning
  - Determinate
  - Efficiently implementable!

Concurrent Constraint Programming

- **Shared store contains (open-ended) set of locations.**
- **Key idea: Accumulate constraints on shared variables.**
  - X=Y, X=1, X > Y+Z, X = cons(Y, Z), 3 in X("cat")
- **Two basic operations (in lieu of Read and Write)**
  - **Tell -- c**: Add **c** to the store
  - **Ask -- if (c) A:** Suspend until the store is strong enough to entail c, then reduce to A.

(Agents) **A**::=

```
c;
if (c)   {A}
A B
{val x:T; A}
```

Use constraints for communication and control between concurrent agents operating on a shared store.

Semantics

Configuration

(Config) `G ::= A,…, A` (multiset of agents)

Reduction Rules

`G, {val x:T; A}` → `G,A` `(x not free in G)`

`G, A B` → `G,A,B`

$G, c_1, \ldots, c_n,$ `if (c) A` → `G,A` $(c_1, \ldots, c_n \vdash c)$

Denotation

Determinate!

[[A]] = **function** mapping initial store to final store (or limit)

Observation: Function is a closure operator (monotone, extensive, idempotent)

Observation: Closure operator representable by a single set (its fixed points). (P(a) is just the least fixed point of P above a.)

Observation: Parallel composition is just set intersection!

No messy interleavings!

Example program: quicksort

```
class Cons[T](h:T, t:List[T])
  implements List[T] {
  def qsort() {
    val x=tail.split(h);
    x.a.qsort()
     .append(Cons(h,x.b.qsort()))
  }
  def split(i:T){T <: Comparable[T]}
   : Pair(List[T], List[T]) {
    val x=t.split(i);
    h < t ?
      Pair(Cons(h,x.a), x.b)
      : Pair(x.a, Cons(h,x.b))
  }
  def append(L:List[T])
     = Cons(h,t.append(L));
  …
}
```

```
class Null[T] implements List[T] {
  def qsort()=this;
  def append(L:List[T])=L;
  def split(i:T)=this;
…
}
```

```
struct Pair[S,T](a:S,b:T) {}
```

Invocation

```
val B:Cons[Int];
A=B.qsort();
B=Cons(1,C);
C=Cons(45,D);
D=Null[Int]();
```

Method invoked with target an unbound promise

Information about target computed incrementally;

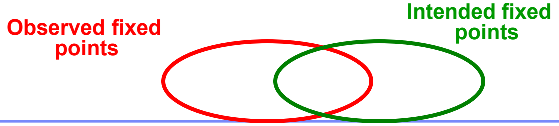
triggers evaluation of qsort body

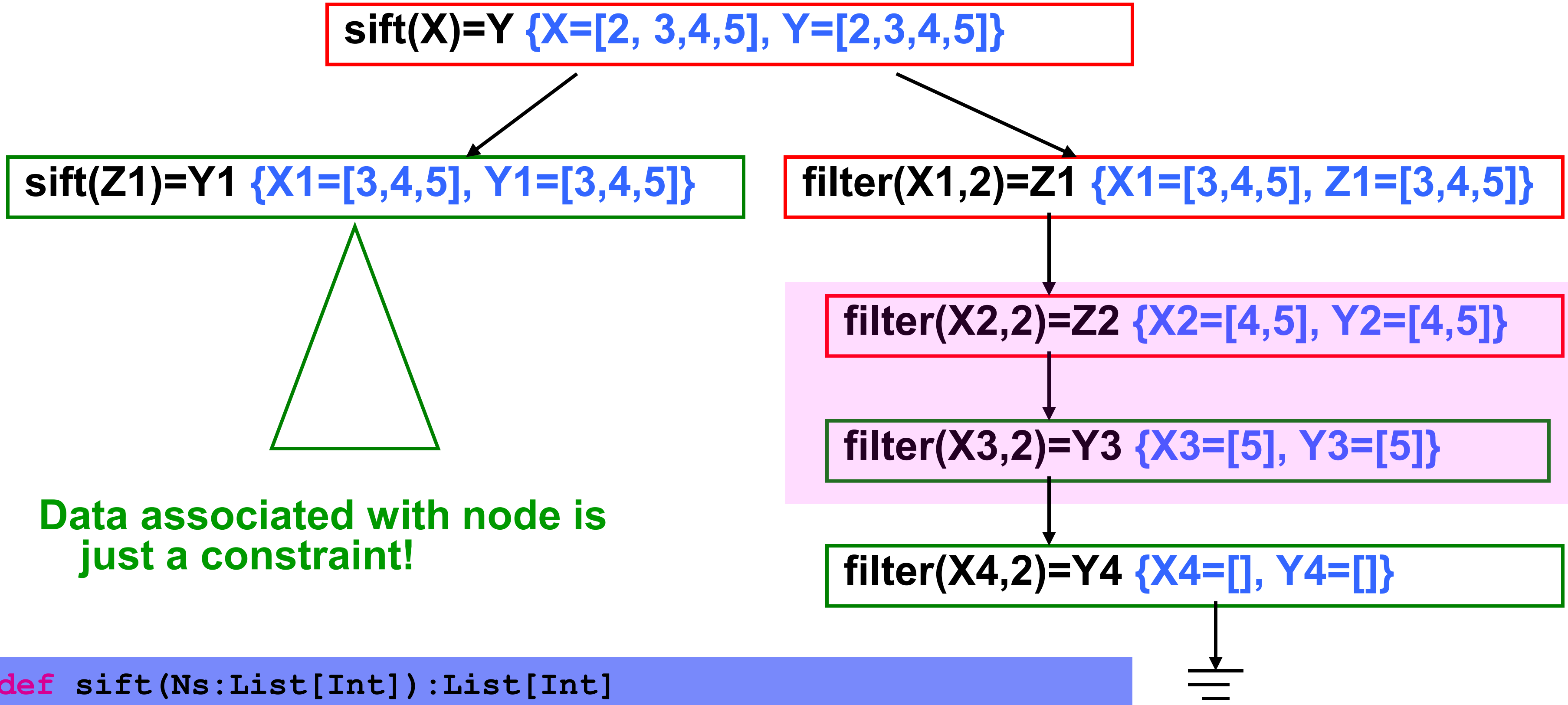
Expressiveness

- **Supports very rich communication patterns**
  - Capturing domain-specific inference rules.
- **Supports mutually recursive processes**
- **Supports dynamic memory allocation ("new")**
- **Subsumes**
  - Concurrent logic programming
  - First-order functional programming
  - Kahn data-flow networks

- **Supports usual concurrent logic programming idioms (Shapiro 83)**
  - "logical variables"
  - Short-circuits for quiescence detection (PODC 88)
  - Difference lists
  - Incomplete messages
  - Streams, trees, arrays, hash-tables
  - … all are refinable, not updatable.

Declarative Debugging of CCP





sift(X)=Y {X=[2, 3,4,5], Y=[2,3,4,5]}

sift(Z1)=Y1 {X1=[3,4,5], Y1=[3,4,5]}

filter(X1,2)=Z1 {X1=[3,4,5], Z1=[3,4,5]}

filter(X2,2)=Z2 {X2=[4,5], Y2=[4,5]}

filter(X3,2)=Y3 {X3=[5], Y3=[5]}

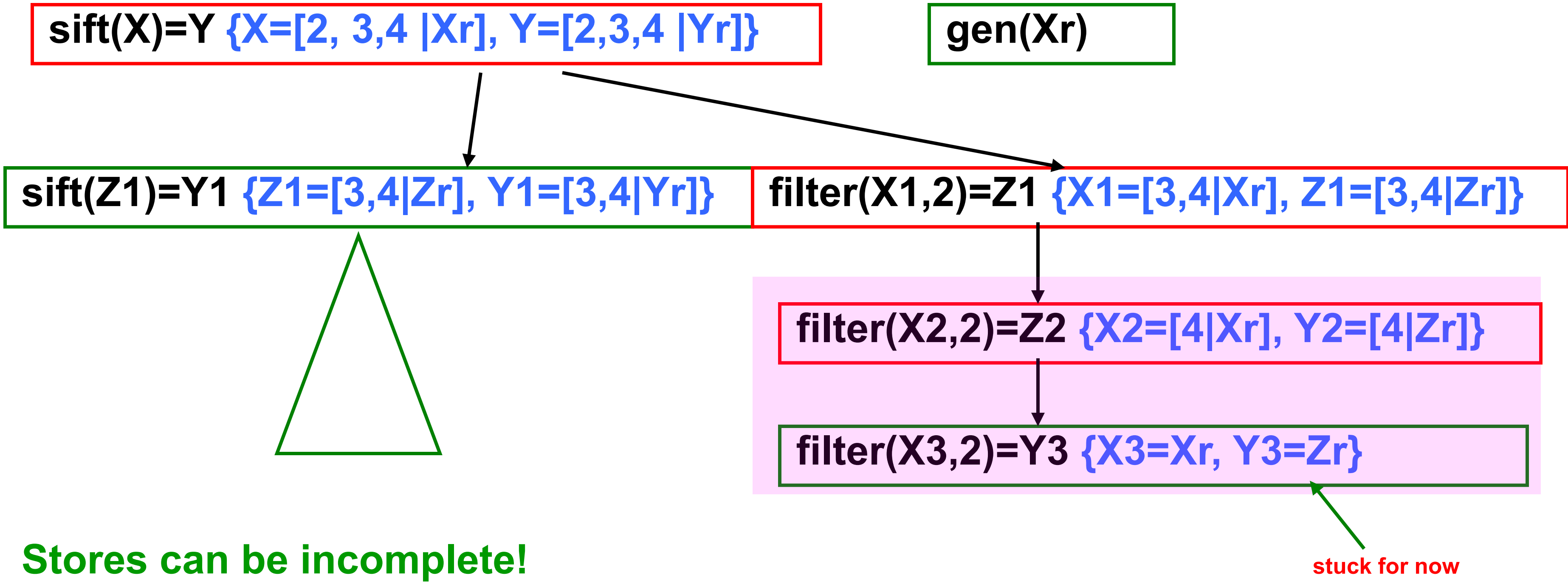
filter(X4,2)=Y4 {X4=[], Y4=[]}

Data associated with node is just a constraint!

```
def sift(Ns:List[Int]):List[Int]
  = Ns.null() ? Null[Int]()
     : Cons(Ns.head, sift(filter(Ns.tail, Ns.head)));
def filter(Ns:List[Int], N:Int):List[Int] {
  = Ns.null() ? Null[Int]()
     : 0==x % N ? Cons(Ns.head, filter(Ns.tail,N))
         : Cons(Ns.head, filter(Ns.tail,N));
```

Live Debugging

Observed fixed points



sift(X)=Y {X=[2, 3,4 |Xr], Y=[2,3,4 |Yr]}

gen(Xr)

sift(Z1)=Y1 {Z1=[3,4|Zr], Y1=[3,4|Yr]}

filter(X1,2)=Z1 {X1=[3,4|Xr], Z1=[3,4|Zr]}

filter(X2,2)=Z2 {X2=[4|Xr], Y2=[4|Zr]}

filter(X3,2)=Y3 {X3=Xr, Y3=Zr}

stuck for now

Stores can be incomplete!

Can debug a subcomputation even with live concurrent agents

```
def sift(Ns:List[Int]):List[Int]
  = Ns.null() ? Null[Int]()
     : Cons(Ns.head, sift(filter(Ns.tail, Ns.head)));
def filter(Ns:List[Int], N:Int):List[Int] {
  = Ns.null() ? Null[Int]()
     : 0==x % N ? Cons(Ns.head, filter(Ns.tail,N))
         : Cons(Ns.head, filter(Ns.tail,N));
```

Default CCP

- `A ::= unless(c) A`
  - Run A, unless c holds **at end**
  - ask c \/ A
  - Leads to nondet behavior
- `unless(c) c;`
  - No behavior
- $\texttt{unless(}c_1\texttt{) }c_2\texttt{; unless(}c_2\texttt{) }c_1\texttt{;}$
  - gives c_1 or c_2

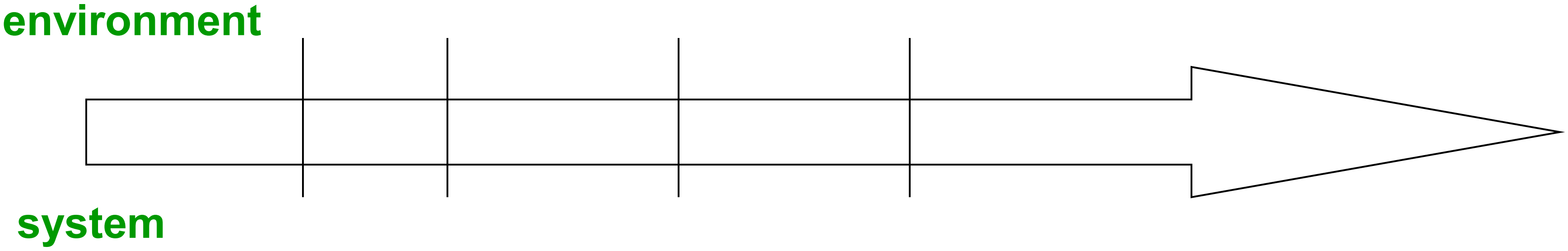
- `unless(c) d;` **: gives d**
- `c; unless(c) d;` **: gives c**

non-monotonicity

- **[A] = set S of pairs (c,d) satisfying**
  - S_d = {c | (c,d) in S} denotes a closure operator.
  - *We still have a simple denotational semantics!*
- **Operational implementation:**
  - Backtracking search
  - Compile-time determinacy analysis (not implemented)
  - Open question:
    - Efficient compile-time analysis (cf causality analysis in Esterel)
    - Use negation as failure

Discrete Timed CCP

Berry's Synchrony Hypothesis



- **Synchronicity principle**
  - System reacts instantaneously to the environment
- **Semantic idea**
  - Run a (bounded) default CCP program at each time point to determine instantaneous response and program for next time instant (resumption)
  - Add: `A ::= next A`
  - No connection between the store at one point and the next.
  - Future cannot affect past.

- **Semantics**
  - Sets of sequences of (pairs of) constraints
  - Non-empty
  - Prefix-closed
  - P after s =d= {e | s.e in P} must be denotation of a Default CC program

- **Determinacy guaranteed if `unless` used only with `next`:**
  - `unless (c) next A;`

Reintroduces "mutation" but in a controlled way – only when the clock ticks!

Hybrid Systems

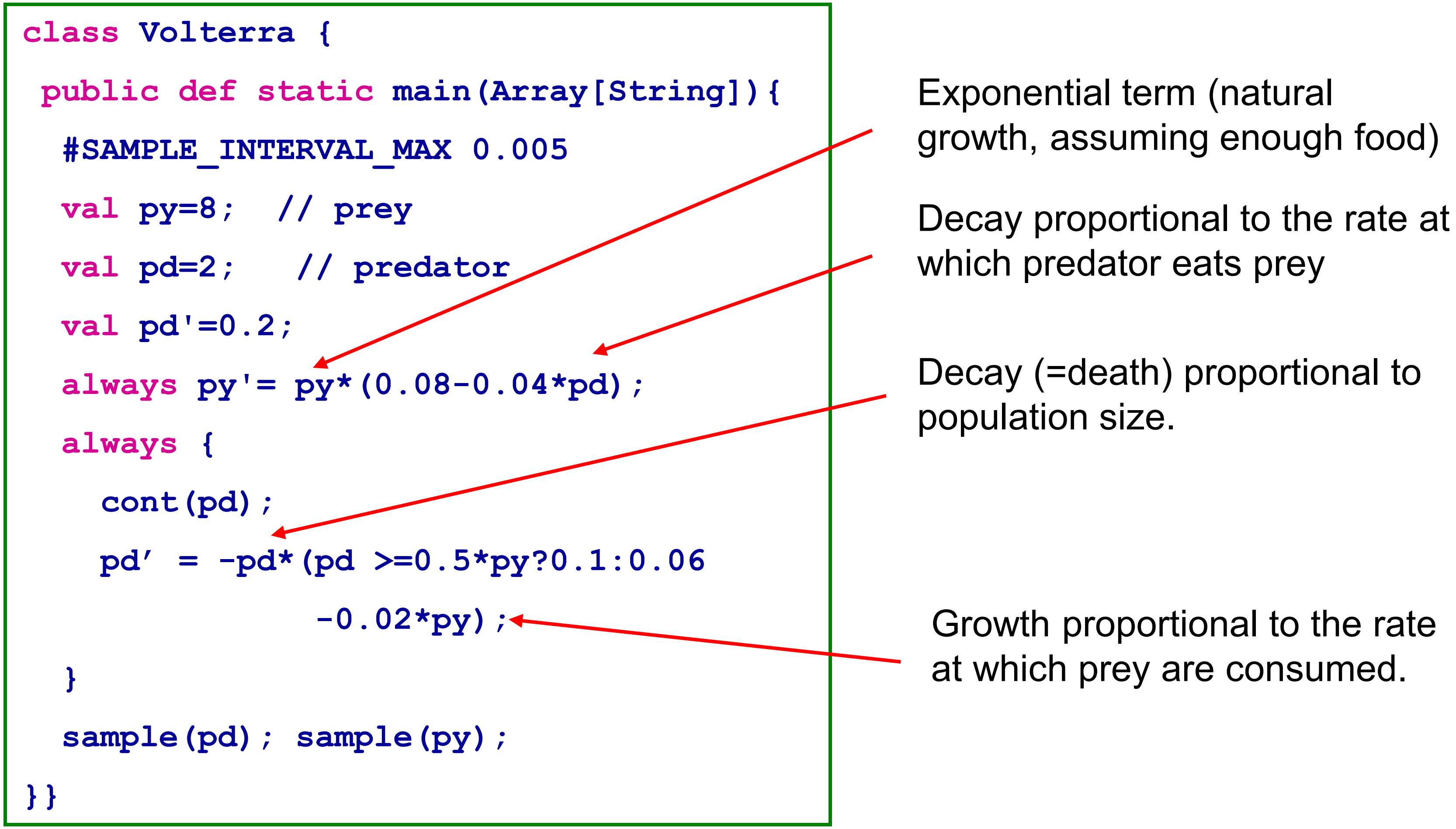
- **Traditional Computer Science**
  - Discrete state, discrete change (assignment)
  - E.g. Turing Machine
  - Brittleness
    - Small error ➔ Major impact
    - Devastating with large code
    - Primary application areas
- **Traditional Mathematics**
  - Continuous Variables (Reals)
  - Smooth state change
    - Mean-value theorem
    - E.g. computing rocket trajectories
  - Robustness in the face of change
  - Stochastic systems (e.g. Brownian motion).
- **Hybrid Systems combine both**
  - Discrete control
  - Continuous state evolution
  - Intuition: Run program at every real value.
    - Approximate by:
      - Discrete change at an instant
      - Continuous change in an interval
- **Primary application areas**
  - Engineering and Control systems
    - Paper transport
    - Autonomous vehicles…
  - Biological Computation.
  - *Programmable Matter?*

Emerged in early 90s in the work of Nerode, Kohn, Alur, Dill, Henzinger…

HCC: Move to Continuous time

- ***No new combinator needed***
  - Constraints are now permitted to vary with time (e.g. `x'=y`)
- **Semantic intuition**
  - Run a Default CC computation at each real time instant, starting with t=0.
  - Evolution of system is piecewise continuous: system evolution alternates between point phase and interval phase.
  - In each phase a Default CC program determines output of that phase and program to be run in next phase.
- **Point phase**
  - Result determines initial conditions for evolution in the subsequent interval phase
- **Interval phase**
  - Any constraints asked of the store recorded as transition conditions.
  - ODE’s integrated to evolve time-dependent variables.
  - Phase ends when any transition condition potentially changes status.
  - (Limit) value of variables at the end of the phase can be used by the next point phase.

Volterra-Lotka model – non-linear differential equations



```
class Volterra {
 public def static main(Array[String]){
  #SAMPLE_INTERVAL_MAX 0.005
  val py=8;  // prey
  val pd=2;   // predator
  val pd'=0.2;
  always py'= py*(0.08-0.04*pd);
  always {
    cont(pd);
    pd′ = -pd*(pd >=0.5*py?0.1:0.06
                -0.02*py);
  }
  sample(pd); sample(py);
}}
```

Exponential term (natural growth, assuming enough food)

Decay proportional to the rate at which predator eats prey

Decay (=death) proportional to population size.

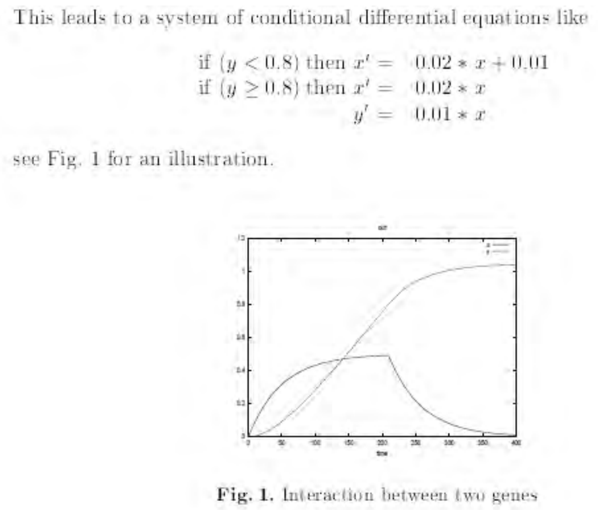
Growth proportional to the rate at which prey are consumed.

Execution introduces adaptive discretization

State dependent rate equations

- **Expression of gene x inhibits expression of gene y; above a certain threshold, gene y inhibits expression of gene x:**

```
if (y < 0.8)
  x'= -0.02*x + 0.01;
if (y >= 0.8) {
  x'=-0.02*x;
  y'=0.01*x;
}
```



This leads to a system of conditional differential equations like

$$\begin{aligned} \text{if } (y < 0.8) \text{ then } x' &= 0.02 * x + 0.01 \\ \text{if } (y \geq 0.8) \text{ then } x' &= 0.02 * x \\ y' &= 0.01 * x \end{aligned}$$

see Fig. 1 for an illustration.

Fig. 1. Interaction between two genes

Bockmayr and Courtois: Modeling biological systems in hybrid concurrent constraint programming

Spatial HCC: Move to continuous space

- **Add** `A::= atOther A`
  - Run A at all *other* points. (`atAll A = A, atOther A`)
  - Constraints may now use partial derivatives.
  - All variables now implicitly depend on space parameters (e.g. x,y,z)
- **Semantic intutions**
  - Computation now uniformly extended across space.
  - At each point, run a Default CC program.
  - Program induces its own discretization of space (into open and closed regions).
- **Programming intuition**
  - Program with vector fields, specifying how they vary across space-time.
- **Programming Matter realization**
  - Atoms represent dense computational grid.
  - Signals represented as memory cells in each Atom
  - Atoms use epidemic algorithms to diffuse signals (possibly with non-zero gradients) across space.
  - Atoms use neighborhood queries to sense local minima
  - Atoms integrate PDEs by using chaotic relaxation (Chazan/Mirankar).
  - Compiler produces FSA for each atom from input program.

Implementation Challenges

- **Need coarsening techniques**
  - Formalism exposes very fine-grained concurrency
  - async for every argument evaluation creates excessive overhead
- **Need analysis to eliminate unnecessary promise creation.**
- **Need efficient implementation of suspension**

- **Implementation can reuse**
  - X10 scheduler
    - Currently fork-join, later work-stealing
  - X10’s concurrent allocator, garbage collector
  - X10’s implementation across multiple nodes

Results should be achievable quickly, building on X10 (e.g. annotations)

Research Agenda

- **Develop "broad" programming framework**
  - Declarative programs (CCP)
  - Fundamentally integrates space and time
  - Compiles to high-performance imperative programs
- **Develop tools that exploit declarative semantics**
  - Correctness at scale
  - Correct by construction
  - Partial programs, sketching
  - Declarative debugging
- **Directed at substantially raising level of programmer/productivity**
  - (cf R, Matlab, … but at scale)
  - "domain" programmer: HPC, machine learning/BA

Background

Selected Bibliography

- **Saraswat, Rinard, Panangaden "Semantics of Concurrent Constraint Programming", POPL 1991**
- **Falaschi, Gabbrielli, Marriott, Palamidessi "Compositional analysis for CCP", LICS 1993**
- **Fromherz "Towards declarative debugging of CCP", 1995**
- **Saraswat, Jagadeesan, Gupta "Timed Default CCP", Journal Symbolic Comp., 1996**
- **de Boer, Gabbrielli, Marchiori, Palamidessi "Proving concurrent constraint programs correct", TOPLAS 1997**
- **Gupta, Jagadeesan, Saraswat "Computing with continuous change", Science Comp Progg. 1998.**
- **Etalli, Gabbrielli, Meo "Transformations of CCP programs", TOPLAS 2001**
- **Falaschi, Olarte, Valencia "Framework for abstract interpretation for Timed CCP", PPDP 09**
- **Gabbrielli, Palamidessi, Valencia "Concurrent and Reactive Constraint Programming", 2010**

Constraint systems

- **Any (intuitionistic, classical) system of partial information**
- **For Ai read as logical formulae, the basic relationship is:**
  - A1,…, An |- A
  - Read as “If each of the A1,…, An hold, then A holds”
- **|- is axiomatized through given rules.**
- **Require conjunction, existential quantification**

A,B,D ::= atomic formulae | A&B |X^A

G ::= multiset of formulae

(Id) A |- A (Id)

(Cut) G |- B G’,B |- D ➔ G,G’ |- D

(Weak) G |- A ➔ G,B |- A

(Dup) G, A, A |- B ➔ G,A |- B

(Xchg) G,A,B,G’ |- D ➔ G,B,A,G’ |- D

(&-R) G,A,B |- D ➔ G, A&B |- D

(&-L) G |- A G|- B ➔ G |- A&B

(^-R) G |- A[t/X] ➔ G |- X^A

(^-L,*) G,A |- D ➔ G,X^A |- D

Constraint system: Examples

- **Gentzen**
  - G |- A iff A in G.
- **Herbrand**
  - uninterpreted first-order terms (labeled, fixed-arity trees)
- **Finite domain**
- **Propositional logic (SAT)**
- **Arithmetic constraints**
  - Naïve, linear, nonlinear
- **Interval arithmetic**

- **Orders**
- **Temporal Intervals**
- **Hash-tables**
- **Arrays**

- **Graphs**

- **Constraint systems (as systems of partial information) are ubiquitous in computer science**
  - Type systems
  - Compiler analysis
  - Symbolic computation
  - Concurrent system analysis

Logic

Proposition: Operational Semantics is complete for constraint entailment. (Saraswat, Lincoln 1994, unpublished)

- **CCP is simply a fragment of first-order logic.**
  - Computation == Deduction
  - Unlike "Logic Programming", CCP employs "forward chaining".

- **RCC (Jagadeesan, Nadathur, Saraswat, FSTTCS 2005)**
  - Unifies and subsumes CCP and LP (forward- and backward-chaining).
  - Provides logical expression for recursive nested guards
    - i.e. "finish"
  - Localized augmentation of programs ("assume-if" reasoning, `(P=>Q)=>R)`
  - Backtracking and search

xcc: CCP in X10

- **Basic idea**
  - Concrete language is just like X10 – classes, inheritance, interfaces, structs, functions, fields, methods, constructors, user-defined operators, type inference etc.
  - No var permitted, no need for atomic, when, finish, async, at.
    - Initially, finish, async, at may be introduced as annotations to permit efficient execution while compiler is being developed.
- **Every variable of type T is initialized with a *promise* of type T.**
  - A promise is a "logical variable" – nothing is known about it.
  - (Herbrand) Two objects are equal iff they are instances of the same class and their corresponding fields are equal.
- **Assignment (=) is re-interpreted as Tell:**
  - $e_1=e_2$ is executed as: evaluate e_1 to get a value v_1, e_2 to get v_2, and equate the two.
- **if (and ? : conditional expression evaluator) suspends until condition evaluates to true or false**
  - if = when, because of monotonicity.
- **`e.m(`e_1`,...,`e_n`)`**
  - e, e_1`,...,`e_n evaluated in parallel
  - Once enough is known about e to determine the class, use dynamic lookup to determine method body
  - Body executed in parallel with arg evaluation
    - Return value is an anonymous promise constrained by return statements.

Can computations deadlock?

- **Yes.**
  - `when(a)` b is canonical deadlocked agent.
  - Intuitively, program quiesces but can produce more when given more.
- **Deadlock is a "natural" state.**
  - Simply means the system has quiesced.
  - If you supply more information, you may get more information back.
  - E.g. almost all interesting programs would deadlock on true.
- **Semantic characterization:**
  - P does not deadlock on input a if all fixed points of P above a are stable.
    - **b >= P(a) implies b in P**
  - Observation: if P does not deadlock on d, then for any b, P(d&b)=P(d)&P(b)

Open problem:

Identify static type system that guarantees deadlock-freedom and permits useful idioms to be expressed.

Declarative Debugging

- **Declarative debugging techniques can be applied to logic programs, functional programs, CCP.**
  - Ueda 98 (CCP)
  - Fromherz 93
  - Falaschi et al ICLP 07

- **Basic idea is to summarize an execution through an execution tree**
  - Node = procedure call
  - Children = calls made in the body.
  - Node associated with some data about subtree, e.g. pair of input/output constraints.

- **Debugging**
  - Query oracle (user, specification) whether data with node is correct.
  - Identify node with incorrect data whose children have correct data …. BUG!

Timed CCP: Basic Results

- **TCC = fragment of first-order linear temporal logic**

- **Rich algebra of defined temporal combinators (cf Esterel):**
  - always A
  - do A watching c
  - whenever c do A
  - time A on c

- **A general combinator can be defined**
  - time A on B: the clock fed to A is determined by (agent) B

- **Discrete timed synchronous programming language with the power of Esterel**
  - present is translated using defaults

- **Proof system**

- **Compilation to automata**

Programming matter

- ***Vijay Saraswat, IBM Research***
- ***Radha Jagadeesan, De Paul University***

- ***May 2006***

Programmable matter

- **Large collection of "computing atoms" (catoms) that can**
  - Compute
  - Communicate locally (wireless)
  - Sense
  - Move
  - Adhere to each other (bond)
  - Change physical/chemical properties based on state

- **cf sensor networks**

- **Desired computations**
  - Form a particular shape
  - Sense a particular shape

How do you compute with 10^6 computers/cubic centimeter?

The computational substrate

- **No shared clock.**
- **No shared gobal coordinate system.**
- **No unique ids (but random variables permitted).**
- **No shared mutable state (shared memory).**
- **Catoms randomly distributed in 3D (2D).**
- **Some small subset are "dead on arrival".**

- **Catoms can sense connections with neighboring catoms and send/receive messages.**
- **Catoms can broadcast locally.**
- **Assume boundary conditions are supplied in some fashion.**
- **Catoms are (re-)programmed by "beaming in" code.**

- **Catoms have limited power?**

Cf Amorphous computing

The programming matter challenge

How do you move from a global description to local actions?

- **What is the programming model for programmable matter?**
- **Global program**
  - Specifies constraints on desired interactions of system with environment.
- **Local program: Catom's view**
  - Specifies how each catom in ensemble initiates/responds to messages received from the environment.

- **Our approach: Program globally, implement locally**
  - Treat programmable matter as *matter*
  - Study how matter "computes"
    - Physics
    - Chemistry
    - Biology – developmental biology
  - Study mathematical descriptions of these processes (continuous space, time, differential eqns, stochasticity)
  - Build programming model on these descriptions
  - Compile such global programs to local catom programs: *"correct" by construction!*

From analysis to programming

Constraint systems

- **Any (intuitionistic, classical) system of partial information**
- **For A_i read as logical formulae, the basic relationship is:**
  - $A_1,\ldots, A_n$ |- A
  - Read as "If each of the $A_1,\ldots, A_n$ hold, then A holds"
- **Require conjunction, existential quantification**

A,B,D ::= atomic formulae | A&B |X^A

G ::= multiset of formulae

(Id) A |- A (Id)

(Cut) G |- B G',B |- D ➔ G,G' |- D

(Weak) G |- A ➔ G,B |- A

(Dup) G, A, A |- B ➔ G,A |- B

(Xchg) G,A,B,G' |- D ➔ G,B,A,G' |- D

(&-R) G,A,B |- D ➔ G, A&B |- D

(&-L) G |- A G|- B ➔ G |- A&B

(^-R) G |- A[t/X] ➔ G |- X^A

(^-L,*) G,A |- D ➔ G,X^A |- D

Saraswat, LICS 91

Constraint system: Examples

- **Gentzen**
- **Herbrand**
  - Lists
- **Finite domain**
- **Propositional logic (SAT)**
- **Arithmetic constraints**
  - Naïve
  - Linear
  - Nonlinear
- **Interval arithmetic**
- **Orders**
- **Temporal Intervals**
- **Hash-tables**
- **Arrays**
- **Graphs**
- **Constraint systems are ubiquitous in computer science**
  - Type systems (checking, inference)
  - Static analysis
  - Symbolic computation
  - Concurrent system analysis

Concurrent Constraint Programming

- **Use constraints for communication and control between concurrent agents operating on a shared store.**
- **Two basic operations**
  - **Tell** c: Add c to the store
  - **Ask** c **then** A: If the store is strong enough to entail c, reduce to A.

(Agents) `A ::= c`

`if (c) A`

`A,B`

`{x:T; A}`

(Config) `G ::= A,…,A`

`G,{x:T;A}` ➔ `G,A` (x not free in G)

`G, if (c) A` ➔ `G,A` (s(G) |- c)

[[A]] = set of fixed points of a closure operator

Operational semantics is complete for logical entailment of constraints.

Saraswat 89; POPL 87, POPL 90, POPL 91

Default CCP

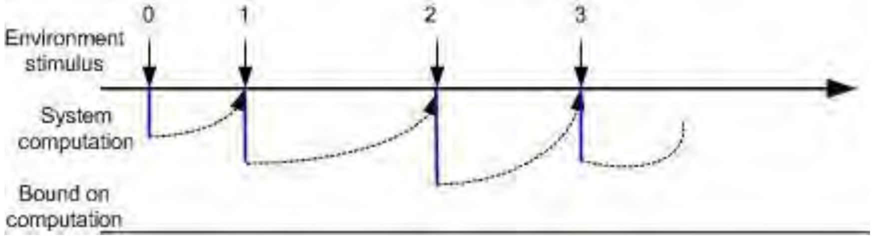
- `A ::= unless(c) A`
  - Run A, unless c holds at end
  - ask c \/ A
  - Leads to nondet behavior
- `unless(c) c`
  - No behavior
- $\texttt{unless}(\texttt{c}_1)\ \texttt{c}_2,\ \ \texttt{unless}(\texttt{c}_2)\ \texttt{c}_1$
  - gives c_1 or c_2
- `unless(c) d` **: gives d**
- `c, unless(c) d` **: gives c**

↑

non-monotonicity

- **[A] = set S of pairs (c,d) satisfying**
  - S_d = {c | (c,d) in S} denotes a closure operator.
  - *We still have a simple denotational semantics!*
- **Operational implementation:**
  - Backtracking search
  - Compile-time determinacy analysis (not implemented)
  - Open question:
    - Efficient compile-time analysis (cf causality analysis in Esterel)
    - Use negation as failure

Discrete Timed CCP (1993)



- **Synchrony principle**
  - System reacts **instantaneously** to the environment
  - Implemented by ensuring computation at each time instant is bounded.
- **Semantic idea**
  - Run a Default CCP program at each time point
  - Add a single new combinator: **A ::= hence A** (run A at every *subsequent* instant.)
  - No connection between the store at one point and the next.
  - Semantics: Sets of sequences of (pairs of) constraints

Proof system

- **The usual temporal combinators can be programmed:**
  - **`always(A) = {A; hence A;}`**
  - **`do A watching c`**
  - **`time A on B:`** the clock fed to A is determined by (agent) B
- **`unless` can be used to retract `hence` constraints**
  - **`next(A) =`**

```
 {X:boolean;
   hence {
    unless(X=true) A;
    hence X=true;
   }
 }
```

Compilation to automata

Hybrid Systems

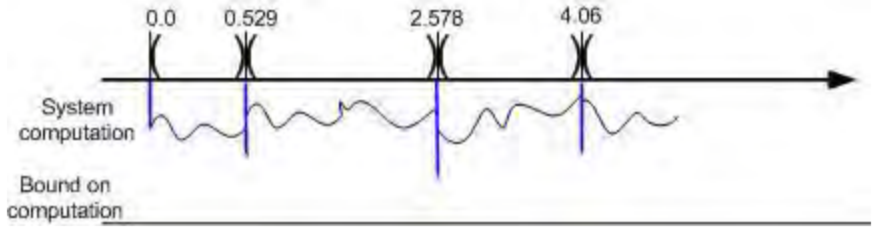
- **Traditional Computer Science**
  - **Discrete state, discrete change (assignment)**
  - **E.g. Turing Machine**
  - **Brittle:**
    - **Small error → major impact**
    - **Devastating with large code!**
- **Traditional Mathematics**
  - **Continuous variables (Reals), with continuous functions (e.g. sum, multiplication).**
  - **Smooth state change**
    - **Mean-value theorem**
    - **e.g. computing rocket trajectories**
  - **Robustness in the face of change**
  - **Stochastic systems (e.g. Brownian motion)**

- **Hybrid Systems combine both**
  - **Discrete control**
  - **Continuous state evolution**
  - **Intuition: Run program at every real value.**
    - **Approximate by:**
      - **Discrete change at an instant**
      - **Continuous change in an interval**
- **Primary application areas**
  - **Engineering and Control systems**
    - **Paper transport**
    - **Autonomous vehicles…**

  - **Biological Computation.**
  - **Programmable Matter**

Emerged in early 90s in the work of Nerode, Kohn, Alur, Dill, Henzinger…

HCC: Move to Continuous time (1995)



- ***No new combinator needed***
  - Constraints are now permitted to vary with time (e.g. `x'=y`)
- **Semantic intuition**
  - Run default CCP at each real time instant, starting with t=0.
  - Evolution of system is piecewise continuous: system evolution alternates between point phase and interval phase.
  - In each phase program determines output of that phase and program to be run in next phase.

- **Point phase**
  - Result determines initial conditions for evolution in the subsequent interval phase and `hence` constraints in effect in subsequent phases.
- **Interval phase**
  - Any constraints asked of the store recorded as transition conditions.
  - ODE's integrated to evolve time-dependent variables.
  - Phase ends when any transition condition potentially changes status.
  - (Limit) value of variables at the end of the phase can be used by the next point phase.

Gupta, Jagadeesan, Saraswat SCP 1998

Systems Biology

- **Work subsumes past work on mathematical modeling in biology:**
  - Hodgkin-Huxley model for neural firing
  - Michaelis-Menten equation for Enzyme Kinetics
  - Gillespie algorithm for Monte-Carlo simulation of stochastic systems.
  - Bifurcation analysis for Xenopus cell cycle
  - Flux balance analysis, metabolic control analysis…

This is not the first time…

- **Why Now?**
  - Exploiting genomic data
  - Scale
    - Across the internet, across space and time.
  - Integration of computational tools
  - Integration of new analysis techniques
  - Collaboration using markup-based interlingua (SBML)
  - Moore’s Law!

Chemical Reactions

- **Cells host thousands of chemical reactions (e.g. citric acid cycle, glycolis…)**
- **Chemical Reaction**
  - $X+Y_0 \xrightarrow{k_0} XY_0$
  - $XY_0 \xrightarrow{k_{-0}} X+Y_0$
- **Law of Mass Action**
  - Rate of reaction is proportional to product of conc of components
  - $[X]' = -k_0[X][Y] + k_{-0}[XY_0]$
  - $[Y]'=[X]'$
  - $[XY]'=k_0[X][Y]-K_{-0}[XY_0]$
- **Conservation of Mass**
- **When multiple reactions, sum mass flows across all sources and sinks to get rate of change.**
- **Same analysis useful for enzyme-catalyzed reactions**
  - Michaelis-Menten kinetics
- **May be simulated**
  - Using “deterministic” means.
  - Using stochastic means (Gillespie algorithm).

At high concentration, species concentration can be modeled as a continuous variable.

Quorum sensing (V. fischeri)

Model due to Alur et al

Cell division: Delta-Notch signaling in X. Laevis

- **Consider cell differentiation in a population of epidermic cells.**
- **Cells arranged in a hexagonal lattice.**
- **Cells interacts concurrently with its neighbors.**
- **Delta and Notch proteins in each cell vary continuously.**
- **Cell can be in one of four states: {Delta, Notch} x {inhibited, expressed}**

- **Experimental Observations:**
  - Delta (Notch) concentrations show typical spike at a threshold level.
  - At equilibrium, cells are in only two states (D or N expressed; other inhibited).

Ghosh, Tomlin: “Lateral inhibition through Delta-Notch signaling: A piece-wise affine hybrid model”, HSCC 2001

Delta-Notch Models

- **Model:**
  - V_D, V_N: concentration of Delta and Notch protein in the cell.
  - U_D, U_N: Delta (Notch) production capacity of cell.
  - U_N=sum_i (neighbors) $V_D(i)$
  - $U_D = -V_N$
  - Parameters:
    - Threshold values: HD,HN
    - Degradation rates: MD, MN
    - Production rates: RD, RN
  - Cell in 1 of 4 states: {D,N} x {Expressed (above), Inhibited (below)}
- **Stochastic variables used to set random initial state.**

if (UN(i,j) < HN) VN'= -MN*VN,

if (UN(i,j)>=HN) VN'=RN-MN*VN,

if (UD(i,j)<HD)   VD'=-MD*VD,

if (UD(i,j)>=HD) VD'=RD-MD*VD,

Results: Simulation confirms observations. Tiwari/Lincoln prove that States 2 and 3 are stable.

Other examples

- **Bouncing ball**
- **Thermostat controller**
- **Square waves**
- **Sine waves…**

- **Paper path model**

- **Aercam model**

Concrete HCC language

- **Arithmetic variables are interval valued.**
- **Arithmetic constraints are non-linear algebraic equations, over +, *, ^, etc.**
- **Users can add own operators as C libraries.**
- **Various combinators translated to basic combinators e.g.**

  do A watching c → execute A, abort it when c becomes true

  when c do A → start A at the first instant when c holds

  wait N do A → start A after N time units

  forall C(X) do A(X) → execute a copy of A for each object X of class C

- **Arithmetic expressions compiled to byte code**
  - Further compiled to machine code.
  - Common sub-expressions are recognized.
- **Copying garbage collector**
  - Speeds up execution
  - Allows snapshotting of state.
- **API from Java/C to use Hybrid cc as a library. System runs on Solaris, Linux, SGI and Windows NT.**

Carlson, Gupta "Hybrid CC with Interval Constraints"

HCC Implementation outline

Constraint techniques

Use constraints to narrow intervals of variables, one variable at a time. Suppose $f(x,y) = 0$.

Indexicals: Rewrite as $x = g(y)$. Set $x \in I \cap g(J)$, where $x \in I$ and $y \in J$. (y can be a vector of variables.)

Interval splitting: If $x \in [a, b]$, use binary search to find min c in $[a,b]$ such that $0 \in f([c,c], J)$, where $y \in J$. Similary determine max such d in $[a,b]$, and set $x \in [c,d]$.

Newton-Raphson: Get min and max roots of $f(x,J) = 0$, where $y \in J$. Set x as above.

Simplex: Given the constraints on x, find its min and max values, and set it as above. Treat non-linear terms as separate variables.

Integration techniques

Treat differential equations as ordinary algebraic equations on variables and their derivatives e.g. $f = m * a''$, $x'' + d*x' + k*x = 0$.

Various integrators are provided --- Euler, 4th order Runge Kutta, 4th order Runge Kutta with adaptive stepsize, Bulirsch-Stoer with polynomial extrapolation. Others can be added if necessary.

Integrators modified to integrate implicit differential equations, over interval valued variables.

Determine points of discrete changes (end of an interval phase) using cubic Hermite interpolation.

Carlson, Gupta "Hybrid CC with Interval Constraints"

Integration of symbolic reasoning

- **Use state of the art constraint solvers**
  - ICS from SRI
  - Shostak combination of theories (SAT, Herbrand, RCF, linear arithmetic over integers).
- **Finite state analysis of hybrid systems**
  - Generate code for HAL
- **Predicate abstraction techniques.**
- **Develop bounded model checking.**
- **Parameter search techniques.**
  - Use/Generate constraints on parameters to rule out portions of the space.
- **Integrate QR work**
  - Qualitative simulation of hybrid systems

Spatial HCC: Move to continuous space

- **Add** `A::= atOther A`
  - Run A at all *other* points. (`atAll A = A, atOther A`)
  - Constraints may now use partial derivatives.
  - All variables now implicitly depend on space parameters (e.g. x,y,z)
- **Semantic intutions**
  - Computation now uniformly extended across space.
  - At each point, run a Default CC program.
  - Program induces its own discretization of space (into open and closed regions).
- **Programming intuition**
  - Program with vector fields, specifying how they vary across space-time.
- **Programming Matter realization**
  - Catoms represent dense computational grid.
  - Signals represented as memory cells in each catom
  - Catoms use epidemic algorithms to diffuse signals (possibly with non-zero gradients) across space.
  - Catoms use neighborhood queries to sense local minima
  - Catoms integrate PDEs by using chaotic relaxation (Chazan/Mirankar).
  - Compiler produces FSA for each catom from input program.

Some basic programming idioms

```
// coord system
R=(0,0,0),
atAll grad(R)=(1,1,1)
// define
at(L) A :: at(R=L) A
at(I:J) A:: at(I<R&R<J) A
```

```
// vibrating 1-d string
u=0, at(R=L)u=0,
at(0<R && R<L)u=f
atAll u''t = c*c*u''x
```

Abbreviation:

`at(boolean b) A ::`

`atAll if (b) A`

`b` may be true at 0 or more points in space.

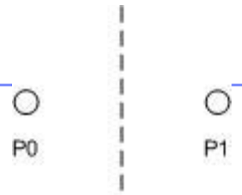
We will also use neighborhood queries:

`min {e | b} (max,…)`

`e` is an expression, `b` a `boolean`

`min` evaluated over a sphere of radius r (execution-time parameter). Also `max`,…

Nagpal's Origami Operator(1): perp



```
agent perp(boolean isP0,
   boolean isP1,
   vec R, // global coord system
   boolean line) {
  at(isP0) {
    vec(2) D0=R, atAll grad(D0)=0.0,
    at(isP1) {
       vec(2) D1=R, atAll grad(D1)=0.0,
       at(norm(D1-D0)<=eps)
         line=true
     }}}
```

Use global coordinate system.

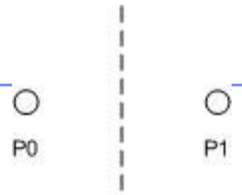
```
agent perp(boolean isP0,
           boolean isP1,

           boolean line) {
  at(isP0) {
    vec(2) D0=0.0, atAll grad(D0)=1.0,
    at(isP1) {
       vec(2) D1=0.0, atAll grad(D1)=1.0,
       at(norm(D1-D0)<= eps)
              line=true
     }}}
```

Use local coordinate systems!

Global coordinate systems can be banned by requiring initial agent is **`atAll A.`**

Nagpal’s Operator(1): perp



```
agent perp(boolean isP0,
           boolean isP1,
           boolean line) {
  at(isP0) {
    vec(2) D0=0.0, atAll grad(D0)=1.0,
    at(isP1) {
       vec(2) D1=0.0, atAll grad(D1)=1.0,
       at(norm(D1-D0) <= eps)
              line=true
     }}}
```

```
agent perp(boolean isP0,
           boolean isP1,
           boolean line) {
  at(isP0) {
    vec(1) D0=0.0,atAll grad(D0)=(1.0,0.0),
    at(isP1) {
       vec(1) D1=0.0,atAll grad(D1)=(1.0,0.0),
       at(norm(D1-D0) <= eps)
              line=true
     }}}
```

Local coordinate system.

Propagates 2-d vectors with unit gradient.

Local *polar* coordinate system.

Propagates scalars with unit radial gradient, zero angular gradient.

Nagpal’s Operator(2): conn

```
agent conn(boolean isP0,
           boolean isP1,
           boolean line) {
  at(isP1) {
    vec(2) D1=0.0, atAll grad(D1)=1.0,
    at(isP0) {
       vec(2) D0=D1, atAll grad(D0)=0.0,
       at(norm(D1.unit-D0.unit)<= eps)
              line=true}}
```

```
agent conn(boolean isP0,
           boolean isP1,
           boolean line) {
  at(isP1) {
    vec(2) D1=0.0,atAll grad(D1)=(1.0,0.0),
    at(isP0) {
      vec(2) D0=0.0,atAll grad(D0)=(1.0,0.0),
      at(D0+D1-min{D0+D1})<= eps)
              line=true}}
```

Local coordinate system.

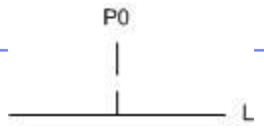
Propagates 2-d vectors with unit gradient.

Local coordinate system.

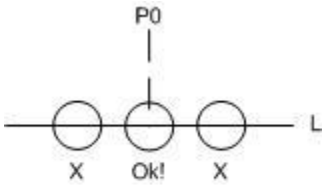
Propagate scalars.

Use neighborhood minima queries.

Nagpal Operator (3): alt



```
agent alt(boolean isP0,
           boolean isLine,
           boolean line, boolean crossing) {
  at(isP0) {
    vec(2) D0=0.0,atAll grad(D0)=(1.0,0.0),
    at(isLine &(D0-min{isLine | D0}<= eps)) {
      crossing=true, atOther crossing=false,
      conn(isP0,crossing,line)}}
```



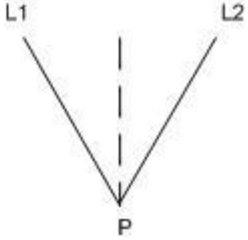
- **Find the point P1 on the line**
  - that is closest to P0
  - in its local neighborhood, considering only points on the line.
- **Draw the line from P0 to P1**

Local coordinate system.

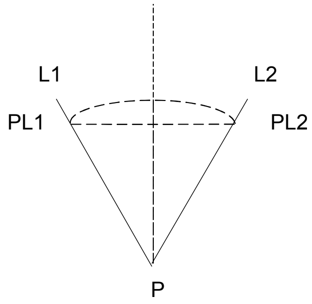
Propagate scalars.

Use *conditional* neighborhood minima queries.

Nagpal Operator(4): Bisection



```
agent bisect(boolean isLine1,
           boolean isLine2,
           boolean line) {
  at(isLine1 & isLine2) {
    boolean isP=true,
    vec(1) P=0.0, atAll grad(P)=(1.0,0.0),
    at(isLine1&(P0-5.0)<eps) {
        boolean isPL1=true,
        at(isLine2&(P0-5.0)<eps) {
          boolean isPL2=true,atOther isPL2=false
          boolean temp,
          conn(isPL1,isPL2,temp),
          alt(isP,temp,line)}}}}
```



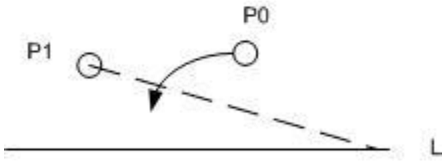
Local coordinate system.

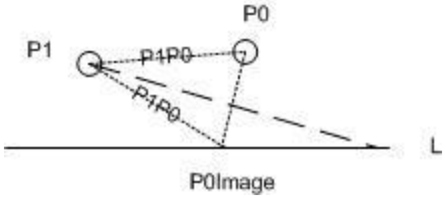
Propagate scalars.

Use other constructions.

Nagpal Operator(5): PontoL

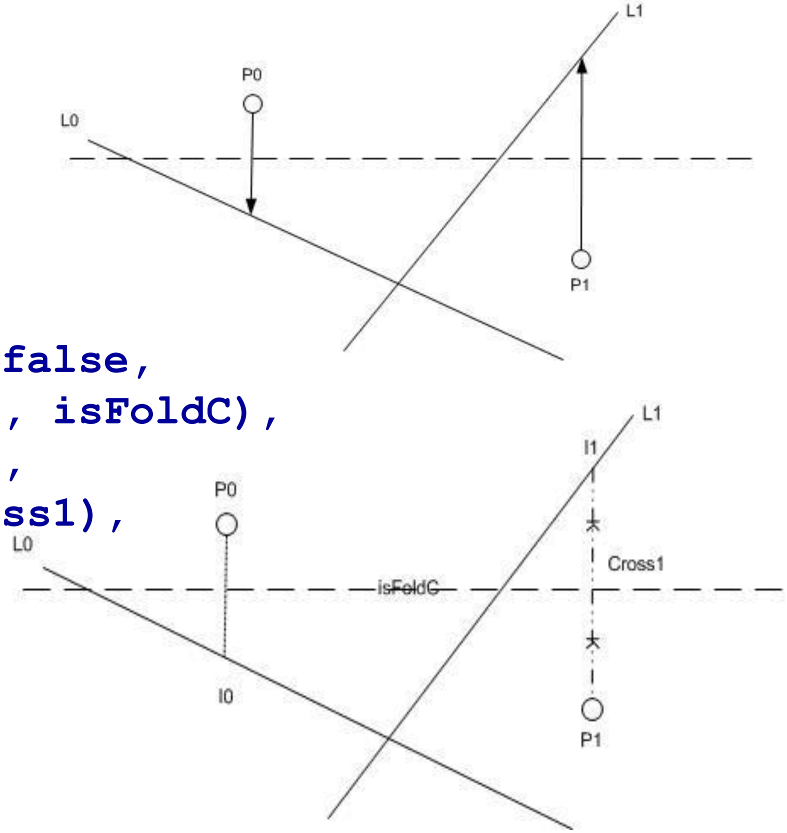
```
agent bisect(boolean isP0,
             boolean isP1,
             boolean isLine,
             boolean line) {
  at(isP0) {
    vec(1) P0=0.0, atall grad(P0)=(1.0,0.0),
    at(isP1) {
        vec(1) P1P0=P0, atAll grad(P1P0)=0.0,
        vec(1) P1=0.0, atAll grad(P1)=(1.0,0.0),
        at(isLine&(P1-P1P0)<eps) {
          boolean isP0Image=true,
          boolean temp, conn(isP0,isP0Image,temp),
          alt(isP1,temp,line)}}}}
```





Nagpal Operator(6): P0P1ontoL0L1

```
agent lineToLines(boolean isP0,
                  boolean isP1,
                  boolean isL0,
                  boolean isL1,
                  boolean isFold) {
    at (isL0) {
        boolean isI0=true, atOther isI0=false,
        boolean isFoldC, perp(isP0, isI0, isFoldC),
        boolean isAlt1, boolean isCross1,
        alt(isP1, isFoldC, isAlt1, isCross1),
         at(isAlt1&isL1) {
             vec(1) orig=0.0,
             atAll grad(orig)=(1.0,0.0),
              at(isCross1) {
                vec(1) K = orig,
                atAll grad(cross1D)=0.0,
                at(isP1&norm(orig-2*K)<eps)
                  atAll isFold = isFoldC
          }}}
```



Flocking

How do u realize this on Progg Matter?

- **Work in progress!**
- **Basic intuitions**
  - Require propagation over space takes time.
  - Dilate time, dilate space.
  - Try establishing computational substrate has, at each point, same velocity of flow (in a particular direction) over time, +/- delta, *with some probability p.*
  - Therefore from each point, sufficiently widely spaced waves are guaranteed to arrive at all other points in sequence.

Conclusion

- **We believe biological system modeling and analysis will be a very productive area for constraint programming and programming languages**

- **Handle continuous/discrete space+time**
- **Handle stochastic descriptions**
- **Handle models varying over many orders of magnitude**
- **Handle symbolic analysis**
- **Handle parallel implementations**

HCC references

- Gupta, Jagadeesan, Saraswat “Computing with Continuous Change”, Science of Computer Programming, Jan 1998, 30 (1—2), pp 3--49
- Saraswat, Jagadeesan, Gupta “Timed Default Concurrent Constraint Programming”, Journal of Symbolic Computation, Nov-Dec1996, 22 (5—6), pp 475-520.
- Gupta, Jagadeesan, Saraswat “Programming in Hybrid Constraint Languages”, Nov 1995, Hybrid Systems II, LNCS 999.
- Alenius, Gupta “Modeling an AERCam: A case study in modeling with concurrent constraint languages”, CP’98 Workshop on Modeling and Constraints, Oct 1998.

Controlling Cell division: The p53-Mdm2 feedback loop

- **1/ $[p53]'=[p53]_0 - [p53]*[Mdm2]*deg - d_{p53}*[p53]$**
- **2/ $[Mdm2]'=p1+p2_{max}*(I\^n)/(K\^n+I\^n)-d_{Mdm2*}[Mdm2]$**
  - I is some intermediary unknown mechanism; induction of [Mdm2] must be steep, n is usually > 10.
  - May be better to use a discontinuous change?
- **3/ $[I]'=a*[p53]-k_{delay}*I$**
  - *This introduces a time delay between the activation of p53 and the induction of Mdm2. There appears to be some hidden "gearing up" mechanism at work.*
- **4/ $a=c_1*sig/(1+c_2*[Mdm2]*[p53])$**
- **5/ $sig'=-r*sig(t)$**
  - Models initial stimulus (signal) which decays rapidly, at a rate determined by repair.
- **6/ $deg=deg_{basal}-[k_{deg}*sig-thresh]$**
- **7/ $thresh'=-k_{damp}*thresh*sig(t=0)$**

Lev Bar-Or, Maya et al "Generation of oscillations by the p53-Mdm2 feedback loop..", 2000

The p53-Mdm2 feedback loop

- **Biologists are interested in:**
  - Dependence of amplitude and width of first wave on different parameters
  - Dependence of waveform on delay parameter.
- **Constraint expressions on parameters that still lead to desired oscillatory waveform would be most useful!**

- **There is a more elaborate model of the kinetics of the G2 DNA damage checkpoint system.**
  - 23 species, rate equations
  - Multiple interacting cycles/pathways/regulatory networks:
    - Signal transduction
    - MPF
    - Cdc25
    - Wee1

Aguda "A quantitative analysis of the kinetics of the G2 DNA damage checkpoint system", 1999

Integration of symbolic reasoning techniques

- **Use state of the art constraint solvers**
  - ICS from SRI
  - Shostak combination of theories (SAT, Herbrand, RCF, linear arithmetic over integers).
- **Finite state analysis of hybrid systems**
  - Generate code for HAL
- **Predicate abstraction techniques.**
- **Develop bounded model checking.**
- **Parameter search techniques.**
  - Use/Generate constraints on parameters to rule out portions of the space.
- **Integrate QR work**
  - Qualitative simulation of hybrid systems