Ubiquity of time … and space

Fixed monitoring networks
- downtown / transportation surveillance
- highway monitoring

Embedding computational artifacts in the real world
- Photocopiers, vehicles

Smarter planet

Vast increasing number of interconnected, embedded networks

Mobile monitoring networks -- cell-phones
- Many new proximity-based services
- On highways?
- Where is my %$ taxi?

Modeling the real world
- Physics
- Chemistry
- Biology

Modeling the real world in computational reality
- Games, virtual worlds, animation

Embedding mobile computational artifacts in the real world
- Robotics

Face-book feeds...
- Has ur house been burgled yet?
Ton of questions…!

- Do I really need to make time and space explicit…?

- What are the space of models for
  - ... space?
    - Geometric?
    - Topological?
  - ... time?
    - Discrete time?
    - Continuous time?
  - For what purpose…?
    - Modeling, simulation, monitoring, planning, synthesis, control, debugging...

- How should stochasticity be handled?

- What are the right programming models for time and space?
  - Languages, constructs, libraries/APIs,
  - Tools – debuggers, static analyzers, ...
(Spatial, Timed) Concurrent Constraint Programming
Concurrent Constraint Programming

- Shared store contains (open-ended) set of locations.

- Key idea: Accumulate constraints on shared variables.
  - $X = Y$, $X = 1$, $X > Y + Z$, $X = \text{cons}(Y, Z)$, 3 in $X$ (“cat”)

- Two basic operations (in lieu of Read and Write)
  - **Tell** -- $c$: Add $c$ to the store
  - **Ask** -- when $(c) A$: Suspend until the store is strong enough to entail $c$, then reduce to $A$.

(Agents) $A := \begin{align*}
  c \\
  \text{when } (c) A \\
  A, B \\
  \{\text{val } x : T, A\}
\end{align*}$

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

Configuration

\[(\text{Config}) \ G : := \ A, \ldots, \ A \text{ (multiset of agents)}\]

Reduction Rules

\[G, \ \{\text{val } x:T, \ A\} \Rightarrow G, A \quad (x \text{ not free in } G)\]

\[G, c_1, \ldots, c_n, \text{when } (c) \ A \Rightarrow G, A \quad (c_1, \ldots, c_n \vdash c)\]

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!
Logic

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

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

- RCC (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
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.
How does indeterminacy arise?

- **Through Atomic Tell:**
  - $\text{tryatomic} \ (c) \ \text{else} \ A$
    - adds $c$ to the store in one step only if resulting store is consistent, else executes $A$
  - $\text{tryatomic} \ (c)$ is shorthand for $\text{tryatomic} \ (c) \ \text{else} \ \text{true}$

```
def choose[T](X:T, Y:T) {
    tryatomic (X=Y);
}
```

- Given multiple agents
  - $(\text{choose} \ (X, 1), \ldots, \text{choose} \ (X, n))$
    - only one will succeed.

- **Through defaults (POPL 95)**
  - $A ::= \text{unless} \ (c) \ A$
    - Run $A$ unless $c$ will hold when the program quiesces.
  - Indeterminate program:
    - $\text{unless} \ (a) \ b, \ \text{unless} \ (b) \ a$
    - can output either $a$ or $b$
    - Also possible that a program may have no output:
      - $\text{unless} \ (a) \ a$

- **Of fundamental interest in expressing instantaneous timeouts in Timed CCP**
Discrete Timed CCP

environment

system

- **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: \( A : : = \text{next } A \)
  - 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 \( s =d= \{ e \mid s.e \text{ in } P \} \) must be denotation of a Default CCP program

- **Determinacy guaranteed if unl ess used only with next:**
  - \( \text{unl ess } (c) \text{ next } A; \)
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
Hybrid Systems

- **Traditional Computer Science**
  - Discrete state, discrete change (assignment)
  - E.g. Turing Machine
  - Britteness
    - 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.
State dependent rate equations

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

  when \((y < 0.8)\)
  \[ x' = -0.02x + 0.01; \]

  when \((y \geq 0.8)\) 
  \[
  \begin{align*}
  x' &= -0.02x, \\
  y' &= 0.01x
  \end{align*}
  \]

**Bockmayr and Courtois: Modeling biological systems in hybrid concurrent constraint programming**
Spatial HCC: Move to continuous space

- **Add** \( A := \text{atOther} \ A \)
  - Run \( A \) at all *other* points.
  - \((\text{atAll} \ A = A, \text{atOther} \ A)\)
  - 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.
Declarative Debugging

- Declarative debugging techniques can be applied to [T]CC.
  - Ueda 98 (CCP)
  - 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 constraint.

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

- Falaschi et al define abstract semantics adequate for data-flow analysis
  - Use over- and under-approximating semantics
  - Use Linear Temporal Logic to specify intended behavior
HCC in X10
What is Asynchronous PGAS?

Two basic ideas: Places and Asynchrony

- Fine grained concurrency
  - async S
- Place-shifting operations
  - at (P) S
- Atomicity
  - atomic S
  - when (c) S
- Ordering
  - finish S
  - clocks
- Global data-structures
  - points, regions, distributions, arrays
  - Supports
clocked final variables (clocked) acc

Immune data: final fields, value type instances

Partitioned Global Address Space (PGAS)
X10: A new Object-Oriented APGAS language

- **Sequential Language: functions + objects**
  - Functions/closures
  - Constrained Types
    - Programmer/compiler more aware of properties of data
    - Catch bugs early, generate better code
  - Generic Types
    - Runtime type reification
  - Integration of objects and structs
    - Inlinable, headerless objects
    - Better memory efficiency

- **Concurrent Language**
  - async, at, finish, atomic, clock
  - Guaranteed multi-place deadlock-free subset
  - Guaranteed determinate subset
    - Safety: Parallel program is equivalent to underlying serial program.
    - Based on effect annotations

- Compiles to Java and C++
- Runs on clusters, BG, P7 IH

Productivity and Performance at Scale
Basic idea
- Concrete language is just like X10 – classes, interfaces, structs, functions, methods, constructors, type inference etc.

For every type $\tau$, it is possible to have a promise of type $\tau$.
- A promise is a “logical variable” – nothing is known about it.
- Promises are first-class. (They can be stored in local variables, fields...)
- Every variable is immutable; it is initialized with a promise.
- Promises and objects can be equated.
- (Herbrand) Two objects are equal iff they are instances of the same class and their corresponding fields are equal.

Assignment ($\equiv$) is re-interpreted as Tell:
- $e_1 \equiv e_2$ is executed as: evaluate $e_1$ to get a value $v_1$, $e_2$ to get $v_2$, and equate the two.

$e.m(e_1, \ldots, e_n)$
- $e, e_1, \ldots, e_n$ evaluated in parallel
- Once enough is known about $e$ to determine the class, use dynamic lookup to determine method body
- Execute body in parallel with args
- Method may return with asyncs left behind

if suspends until condition evaluates to true or false (if = when, because of monotonicity).
No need for finish.
No need for explicit async.
(Use $C(e_1, \ldots, e_n)$ instead of $new\ C(e_1, \ldots, e_n)$ to create an object.)
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
Research Agenda

- Develop “broad” programming framework
  - Declarative programs (CCP)
  - Fundamentally integrates space and time
  - High-performance imperative programs
  - w/ support for provenance, multiple views

- 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
Ton of questions…!

- Do I really need to make time and space explicit…?

- What are the space of models for
  - … space?
    - Geometric?
    - Topological?
  - … time?
    - Discrete time?
    - Continuous time?
  - For what purpose…?
    - Modeling, simulation, monitoring, planning, synthesis, control, debugging…

- How should stochasticity be handled?

- What are the right programming models for time and space?
  - Languages, constructs, libraries/APIs,
  - Tools – debuggers, static analyzers, …
Background
“concurrent constraint programming”

References

The semantic foundations of concurrent constraint programming - psu.edu [PDF]
VA Saraswat, M Rinard, P Panangaden - Proceedings of the 18th ACM SIGPLAN-SIGACT ... 1991 - portal.acm.org
Semantic foundations of concurrent constraint programming Vijay A. Saraswat, Xerox
PARC ... Abstract Concurrent constraint programming [Sar99, SR99] is a sim: ...
Cited by: 1061 - Related articles - Library Search - All 28 versions

Concurrent constraint programming languages
V Saraswat - Dissertation Abstracts International Part B: Science 1990 - csa.com
Concurrent constraint programming languages, ..., INT. PT. B: SCI & ENG.], 50.1212, 1990.
This thesis develops the framework of concurrent constraint programming ...
Cited by: 365 - Related articles - All 2 versions

Concurrent Constraint Programming
U di Genova - Springer
(3). Communication and synchronization are achieved by means of two kinds of actions: Ask (originally introduced by Maher in [9]), which checks whether a store entails a certain constraint, and Tell, which adds a constraint to ...
Related articles

Concurrent constraint programming
C Palamidessi - Language implementation and logic programming: 8th ..., 1994 - books.google.com
Concurrent Constraint Programming Catuscia Palamidessi Dipartimento di Informatica e Scienze dell'Informazione (DISI) Universita di Genova via Benedetto XV 3, 16132, Genova, Italy catuscia@di.unige.it Abstract In ...
Cited by: 3 - Related articles - EL Direct - All 4 versions

Foundations of timed concurrent constraint programming
V Saraswat, R Jagadeesan, V Gupta - Proc. of the Ninth Annual IEEE Symposium on Logic ..., 1994 - Citeseer
We develop a model for timed, reactive computation by extending the asynchronous, untimed concurrent constraint programming model in a simple and uniform way. In the spirit of process algebras, we develop some combinators ...
Cited by: 125 - Related articles - View as HTML - All 10 versions

Oz Explorer: A visual constraint programming tool - psu.edu [PDF]
C Schulte - Lecture Notes in Computer Science, 1996 - Springer
... 6028). References 1. Martin Henz, Gerit Smolka, and JStg Wirtz.
Object-oriented concurrent constraint programming in Oz. In Vjay ...
Cited by: 124 - Related articles - EL Direct - All 12 versions

IBM Research

© 2009 IBM Corporation
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**
  - \( G \vdash A \) 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
Example program: quicksort

class Cons[T](h: T, t: List[T])
  implements List[T] {
    def head() = h;
    def tail() = t;
    def qsort() {
      Pair(S, L) = tail.split(h);
      return
      S.qsort().append(L.qsort().pre(h));
    }
    def split(i: T) { T <: Comparable[T] } {
      Pair(S, L) = t.split(i);
      return h < t ?
      Pair(Cons(h, S), L)
      : Pair(S, Cons(h, L));
    }
  }

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

Invocation

A=B.qsort(),
B=Cons(1, C),
C=Cons(45, D),
D=Null[1nt]()
Example program: Insertion sort

```scala
class Cons[T](h: T, t: List[T])
  implements List[T] {
    def isort() {T<:Comparable[T]} =
      t.isort().insert(h);

    def insert(x: T){T<:Comparable[T]} =
      (x <= h)
        ? Cons(t, this)
        : Cons(h, t.insert(x));
  }

class Null[T] implements List[T] {
  def isort() = this;
  def insert(t: T) = Cons(t, this);
  ...
}
```

Invocation

```scala```
A=qs(B),
B=Cons(1, C),
C=Cons(45, D),
D=Null[Int]()```
```
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.
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,...
Conclusion

- CCP offers a very rich framework for concurrent, determinate, domain-specific programming.
  - Built over arbitrary domain-specific constraint system
  - (T,H)CC handle discrete and continuous time.
- Substantial theory developed over the last 20 years
  - Notably: semantics, proof systems, declarative debugging

- xcc proposal being developed to implement [T]CCP framework in X10.
  - Inherit X10 type system, object structure
  - Uses X10 support for fine-grained concurrency, concurrent garbage-collection

Declarative concurrent programming for the application programmer