Concurrent Logic/Constraint Programming: The Next 10 Years

Kazunori Ueda
Waseda University
ueda@ueda.info.waseda.ac.jp

Two Approaches to Addressing Novel Applications

- Synthetic
  - More expressive power
  - Integration of features
- Analytic
  - Identifying smaller fragments of LP with nice and useful properties
    cf. Turing machines vs. pushdown automata
  - Separation prior to integration

Grand Challenges

- A "λ-calculus" in concurrency field
  cf. *X*-calculus (calculus of *X*)
  X: CS, pi, action, join, gamma, ambient, . . .
- Common platform for non-conventional computations (parallel, distributed, embedded, real-time, mobile, . . .)
- Type systems (in the broadest sense) and frameworks of analysis for both logical and physical properties

LP vs. Concurrent LP

- Concurrent LP = LP + committed choice
  = LP – completeness

???

Choice is essential for specifying arbitration, changes denotational semantics drastically, but otherwise . . .

LP vs. Concurrent LP

- Concurrent LP
  - = LP + directionality (of dataflow)
  - = Logic + embedded concurrency control
- Moded Concurrent LP / CCP:

  ask + tell + strong moding
- Can/should share more interest with (I)LP

Example: Parallel/Network Programming

- Parallel and distributed computing are the difficult areas where we need good models and methodologies to build large applications quickly.
- A good chance for us to demonstrate the power of simple and "usable" languages with an appropriate level of abstraction.
  - – should be simple and accessible to network programmers; otherwise Java will do!

Example: Parallel/Network Programming

- Still done with "classical" constructs
  - – mutex, serialization, monitors, . . .
  - – cf. Obliq, Java, . . .
- . . . or with APIs (cf. MPI)
- Suffering from low-level details
  - – e.g., Unix sockets
- Far from being "provably correct"

Example: Network Applications Need to Deal with:

- Physical locations (nodes, sites)
- Resources
  - – space (heap usage)
  - – time and stack usage
- Security
  - – high-level: safety of comm. protocols etc.
  - – low-level: leave to Java's bytecode verifier?
- Transmission of various entities
- Various patterns of communication

GHC and KL1: Brief History
(cf. CACM March 1993 issue)

- 1983 Concurrent Prolog, PARLOG
- 1983-84 big controversy (in ICOT) on LP vs. concurrent LP for parallel KIP systems
- 1984 Guarded Horn Clauses designed
- 1985 first paper (ICOT TR103, LNCS 221)
- 1985 GHC-to-Prolog compiler (SLP'85)
- 1985-86 subsetting to Flat GHC
- 1986 Prolog-to-GHC compiler (ICLP'86)

Brief History (cont'd)

- 1988 Strand (→ PCN → CC++)
- 1988(?) PIMOS operating system
- 1988 unfold/fold transformation and transaction-based semantics (FGCS'88)
- 1989 Concurrent Constraint Programming (Saraswat)
- 1989 atomic vs. eventual tell discussed
- 1989 message-oriented impl. (LPC'89)

Brief History (cont'd)

- 1987 MRB (1-bit RC) scheme (ICLP'87)
- 1987 // impl. of Flat GHC on Multi-PSI v1 (6 PEs, ICLP'87)
- 1987 book on GHC (in Japanese)
- 1987 ALPS (Maher, ICLP'87)
- 1987-1988 KL1 (with *shoen*, vectors, MRB)
- 1988 // impl. of KL1 on Multi-PSI v2 (64 PEs, FGCS'88, ICLP'89)

Brief History (cont'd)

- 1990 Moded Flat GHC and constraint-based analysis (ICLP'90)
- 1990(?) MGTP-on-KL1 project started
- 1990 first structural OS (InfoJapan'90)
- 1990 Janus (NACLP'90)
- 1990 Computer J. paper on GHC + KL1
- 1991 denotational semantics of CCP (POPL'91)

Brief History (cont'd)

- 1991 AKL (→ Oz → Oz2 → Oz3) (ILPS'91)
- 1992 // impl. of KL1 on PIM/m and PIM/p (FGCS'92)
- 1992 various parallel applications written in KL1 including OS, biology, CAD, NL, law, automated deduction, . . .
- 1992 message-oriented // impl. of Moded Flat GHC on SMP (FGCS'92)

Brief History (cont'd)

- 1995 constraint-based mode systems in practice (ICLP'95, PSLS'95)
- 1996 klint v1 (mode analyzer for/in KL1)
- 1996 constraint-based error diagnosis, theory and practice (JICSLP'96)
- 1997 kima v1 (diagnoser) based on klint
- 1997 constraint-based error correction

Brief History (cont'd)

- 1992 KLIC (KL1-to-C compiler) desinged (KL1 without *shoen* or MRB)
- 1992 MGTP solved an open problem (IJCAI'93 award)
- 1994 proof system for CCP (POPL'94)
- 1994 KLIC paper (PLILP'94)
- 1994 Moded Flat GHC in detail (NGC)
- 1994 Toontalk (ICLP'95)

Brief History (cont'd)

- 1998 klint v2 with linearity (static MRB) & type analysis
- 1998 NSTO analysis for Moded Flat GHC

Guarded Horn Clauses and KL1

- Weakest Concurrent Constraint Language
  - ask + eventual tell (asynchronous)
  - parallel composition
  - hiding
  - nondeterministic choice
- A realistic language as well as a model
  - value passing
  - data structures (cf. CCS, CSP, . . .)

Logical Variables as Communication Channels

- Data- and demand-driven communication
- Messages with reply boxes
- First-class channels (encoded as lists or difference lists
- Replicable read-only data
- Implicit redirection across sites

Guarded Horn Clauses and KL1

- Evolving process structures (since 1985)
- Physical locations (KL1)
- Object identity (by logical variables)
- I/O completely within the basic framework
- Read/write capabilities (with strong moding)
- Resource-conscious programming (with linearity)
- Scope extrusion ("method calls" encoded as messages)

Guarded Horn Clauses as CCP

- ". . . it is quite natural to view a GHC program in terms of binding information and the agents that observe and generate it."
- "In general, a goal can be viewed as a process that observes input bindings and generates output bindings according to them. Observation and generation of bindings are the basis of computation and communication in our model." — *ICOT TR-208 (1986)*

GHC after 13 years

- The simplest fragment of CCP turned out to be surprisingly versatile, after heated discussions and programming experiences.
- As conjectured in 1985 (LNCS 221), GHC as the weakest fragment of CCP has been (d)evolving by featuring static constructs.
  - static constructs added: mode systems
  - operational constructs added: @node( ) and priorities

GHC after 13 years

- Yet to see what additional features are really necessary, and why
  - Example: higher-order constructs
    - "design pattern" programming
    - object encoding
- Pure CCP or impure CCP?
  - cf. Oz approach (ports, cells, higher-order, etc.)

GHC after 13 years

- Strong moding (1990, ICLP) ensures some aspects of security by assigning a capability/ polarity to each variable occurrence and each position of data structures.
  - write capability can't be duplicated or discarded
  - read, non-linear capability can be duplicated
  - linearity avoids distributed GC
  - unification (constraint solving) degenerates to assignment

Working Example of Network Applications

- 1996 KLIC Programming Contest
  - KLIC = KL1 (GHC) implementation on Unix (http://www.icot.or.jp)
  - Submitted programs included
    - Web server totally written in KL1, and
    - Web browser with most features.
  - Call for Participation: 1998 KLIC Programming Contest (http://www.icot.or.jp)

Some Failures and Problems

- Misleading names
  - Concurrent LP: Sounds like an incomplete variant of LP (worse: committed-choice LP)
  - CCP: Liable to forget its prehistory (<1987)

  Concurrent LP *is* CCP.
- Can very easily be forgotten by LP, concurrency and constraint communities

Some Failures and Problems

- Good textbooks and tutorial materials yet to be published
- Few research groups (except semantics)
  - Oz (DFKI + SICS + . . .)
  - GHC/KLIC (AITEC = former ICOT)
  - many people “graduated” too early

Some Failures and Problems

- Shortage of communication with neighboring communities (functional, OO, . . .)
  - e.g., declarative arrays, program analysis
- Simple and general, but looks a bit too abstract — idioms should be encoded (cf. objects, messages, channels, . . .).

Challenges to share with proponents of other declarative paradigms

- How can we program XXX in our formalisms?
  - Dynamic data structures (e.g., cyclic graphs)
  - MUD and virtual reality
  - Teleconferencing
  - Forms
  - Live Access Counters
  - . . .

LP and Concurrent LP/CCP

- Targetted (currently) at different levels:
  - LP: KR, reasoning, search, etc.
  - Concurrent LP: simple model for concurrency and communication
  - CCP: unified model for reactive systems and infrastructures for reactive agents
  - Should be very carefully 'integrated'
- However, they still have much in common and can benefit from each other!

Conclusions (2)

- Constraint-based static systems can make CCP a simple, powerful, and safe language for
  - parallel computing,
  - distributed computing,
  - real-time computing, and
  - high-performance computing,

  and give us strong support for programming.

  cf. untyped vs. typed λ-calculus

Conclusions (1)

- CCP without static systems has been a simple and elegant formalism of concurrency, . . .

  . . . and at the same time it has been a stable, full-fledged programming language.
  - cf. other formalisms of concurrency

PART II

Potentialities of Constraint-Based Program Analysis

Can a machine debug your program?

```
append([ ],     Y,Z  ) :- true | Y=Z.
append([A|Y],Y,Z0) :- true |
                          Z0=[A|Z], append(X,Y,Z).
```

- ">90% correct"
- cf. Ill-formed sentences in NL processing

Can a machine debug your program without specifications?

```
append([ ],     Y,Z  ) :- true | Y=Z.
append([A|Y],Y,Z0) :- true |
                          Z0=[A|Z], append(X,Y,Z).
```

- Mode analyzer suspects that X is *the* reason.
- The problem can be fixed by replacing X or making X have more occurrences.
- The debugger searches well-moded mutations. Typing doesn't help in this case.

Can a machine debug your program without specifications?

```
append([ ],     Y,Z  ) :- true | Y=Z.
append([A|Y],Y,Z0) :- true |
                          Z0=[A|Z], append(X,Y,Z).
```

- Non-well-moded, under the assumption of "cooperative communication."
- Mode analyzer finds a minimal inconsistent set of mode constraints, which suspects X in the recursive call and the first Y in the head.

Can a machine debug your program without specifications?

```
append([ ],     Y,Z  ) :- true | Y=Z.
append([A|Y],Y,Z0) :- true |
                          Z0=[A|Z], append(X,Y,Z).
```

- The debugger finds 6 alternatives, but prefers 'generic' programs and propose:

```
append([A|X],Y,Z0) :- true |
                          Z0=[A|Z], append(X,Y,Z).
```

- We are happy if the system proposes the intended program and nothing else . . .

Can a machine debug your program without specifications?

```
append([ ],     Y,Z  ) :- true | Y=Z.
append([A|Y],Y,Z0) :- true |
                          Z0=[A|Z], append(X,Y,Z).
```

. . . but it proposes one more alternative:

```
append([A|Y],X,Z0) :- true |
                          Z0=[A|Z], append(X,Y,Z).
```

- Fine, it's not append but does something meaningful (unlike many other junks)!

References

- Cho, K. and Ueda, K.:
  Diagnosing Non-Well-Moded Concurrent Logic Programs (JICSLP'96).
- Ueda, K., Ajiro, Y. and Cho, K.:
  Error-correcting Source Code (submitted to CP'98)