The Exciting Time and Hard-Won Lessons of the Fifth Generation Computer Project

Logic/Constraint Programming and Concurrency

Kazunori Ueda, Waseda University

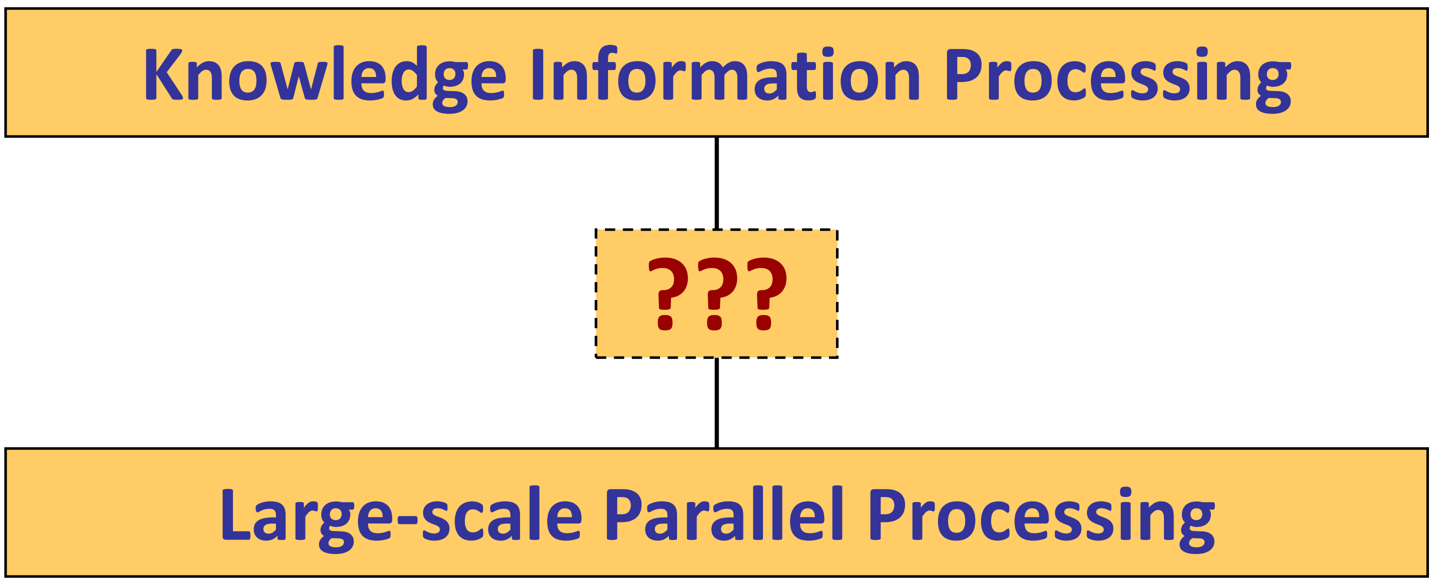
(Formerly with ICOT and NEC (1983-1993))

FLOPS 2016, March 4, 2016

Overview of the FGCS Project

The FGCS Project (1982–1993)

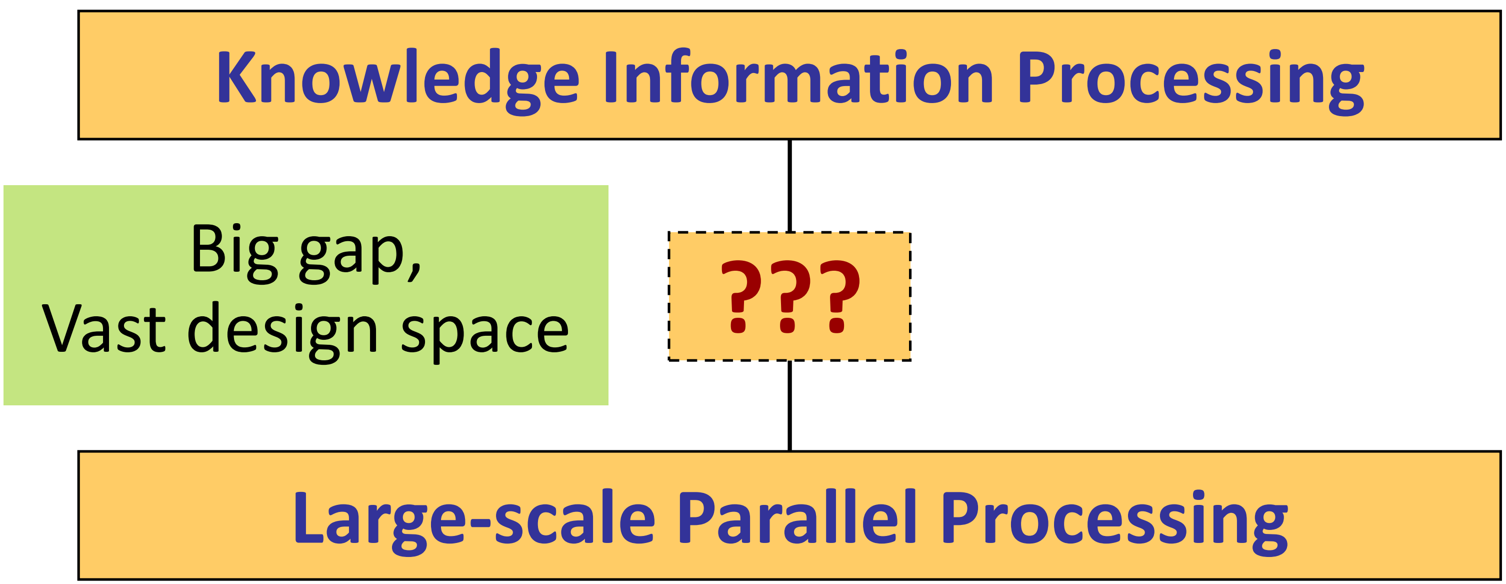
- **Fifth Generation Computer Systems** project (54BJPY)
- **The Challenge:** Bridging Knowledge Information Processing and Parallel Processing



cf. Computer architecture as the hardware/software interface

The FGCS Project (1982–1993)

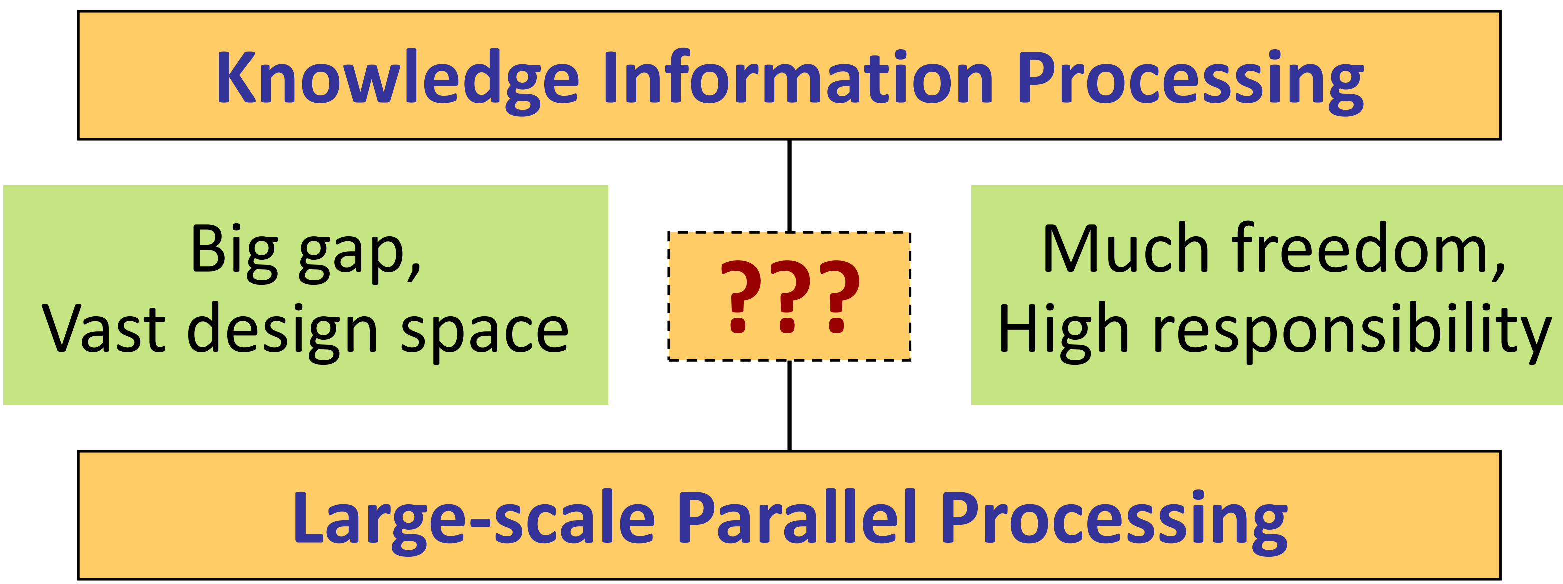
- **Fifth Generation Computer Systems** project (54BJPY)
- **The Challenge:** Bridging Knowledge Information Processing and Parallel Processing



cf. Computer architecture as the hardware/software interface

The FGCS Project (1982–1993)

- **Fifth Generation Computer Systems** project (54BJPY)
- **The Challenge:** Bridging Knowledge Information Processing and Parallel Processing

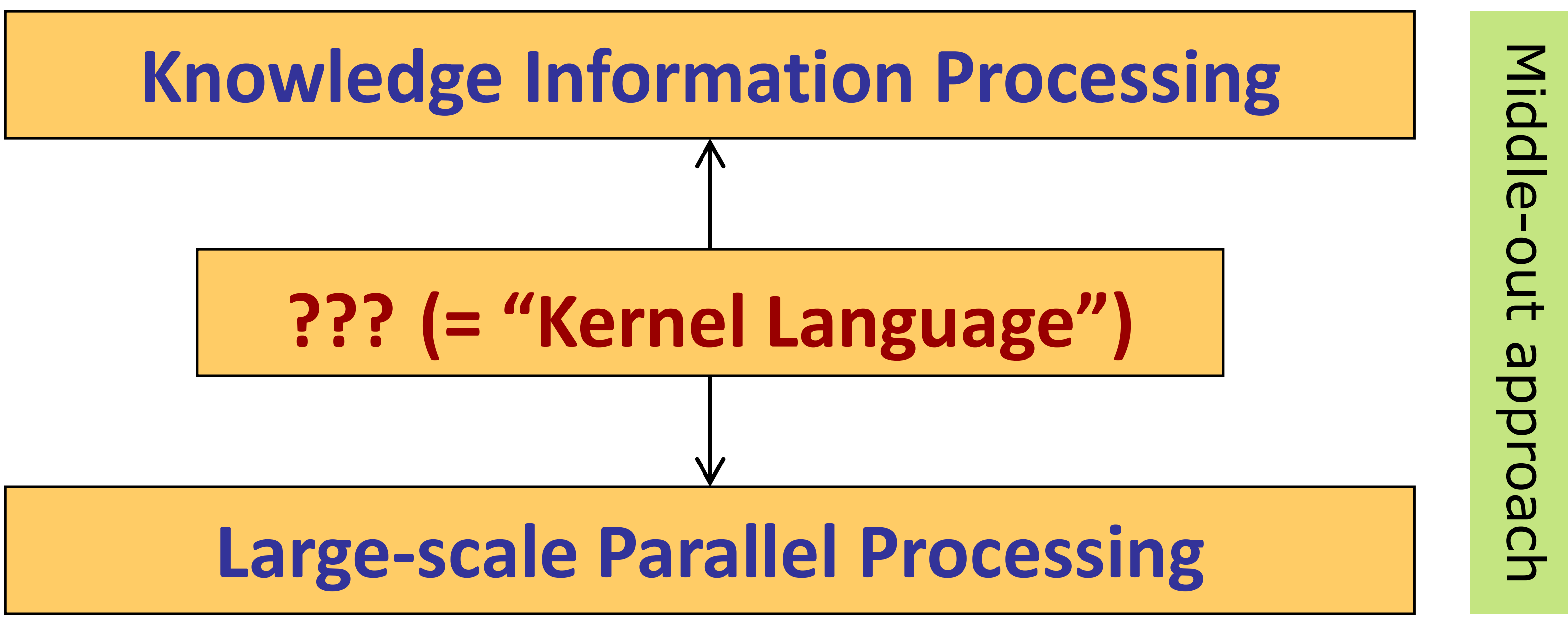


cf. Computer architecture as the hardware/software interface

The FGCS Project (1982–1993)

- *Many hypes and myths*, but its theme was:

  "to develop a methodology and a computer system that bridges Knowledge Information Processing and Parallel Processing"



Who conducted R&D, and where

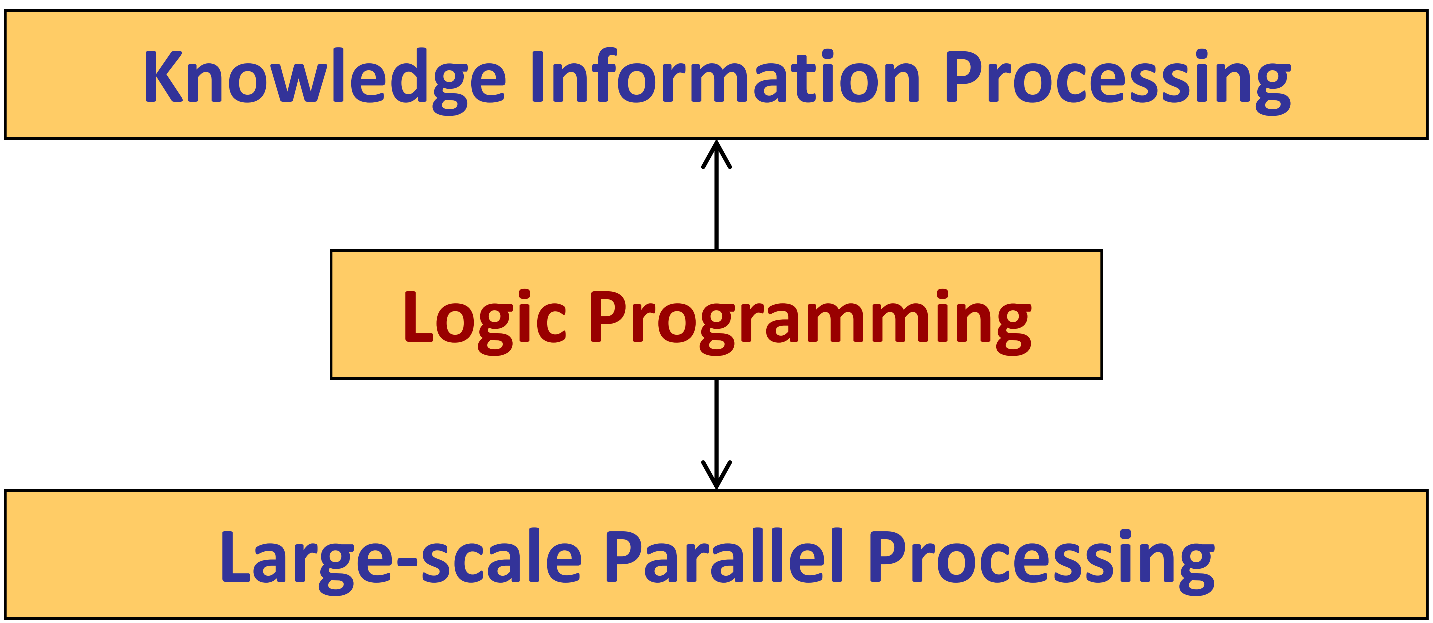
ICOT Research Center
21st floor (+ machine rooms etc.)

ICOT Colleagues reunion in Jan. 2014

Starting with a Working Hypothesis

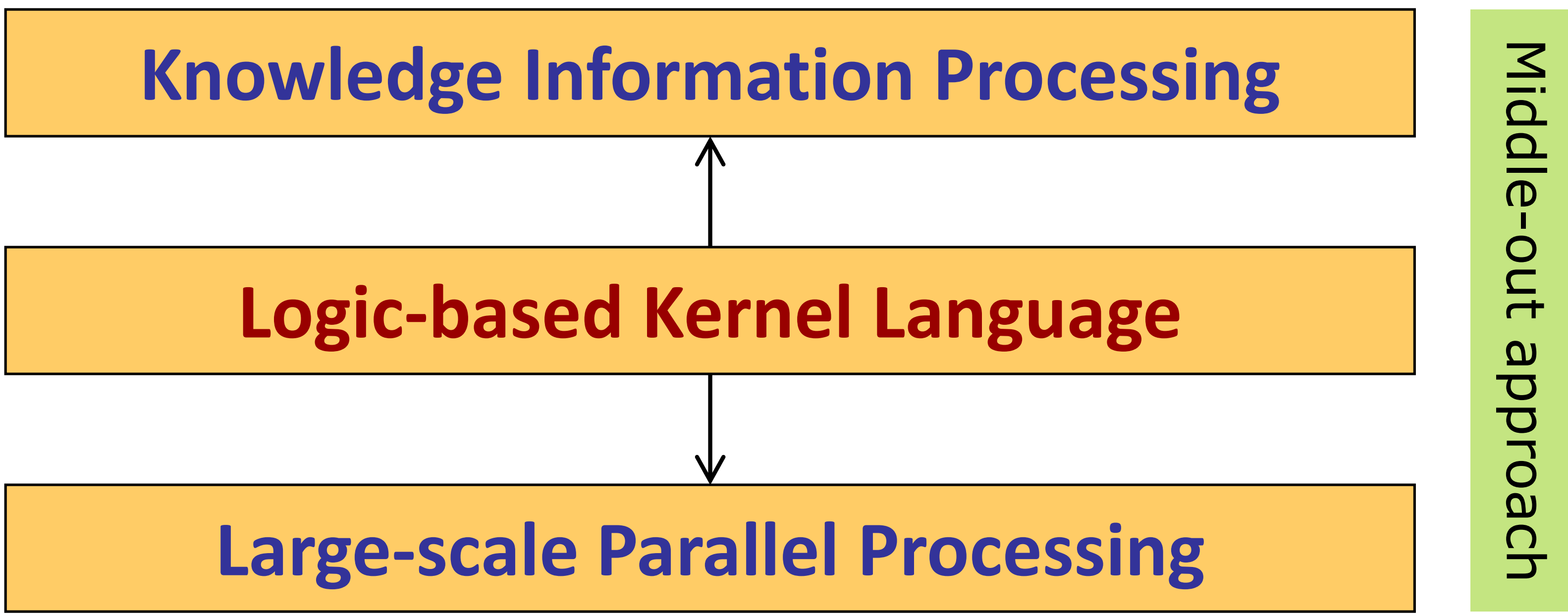
- Logic Programming (in a broad sense) chosen as the working hypothesis
  - for two reasons

Research question here



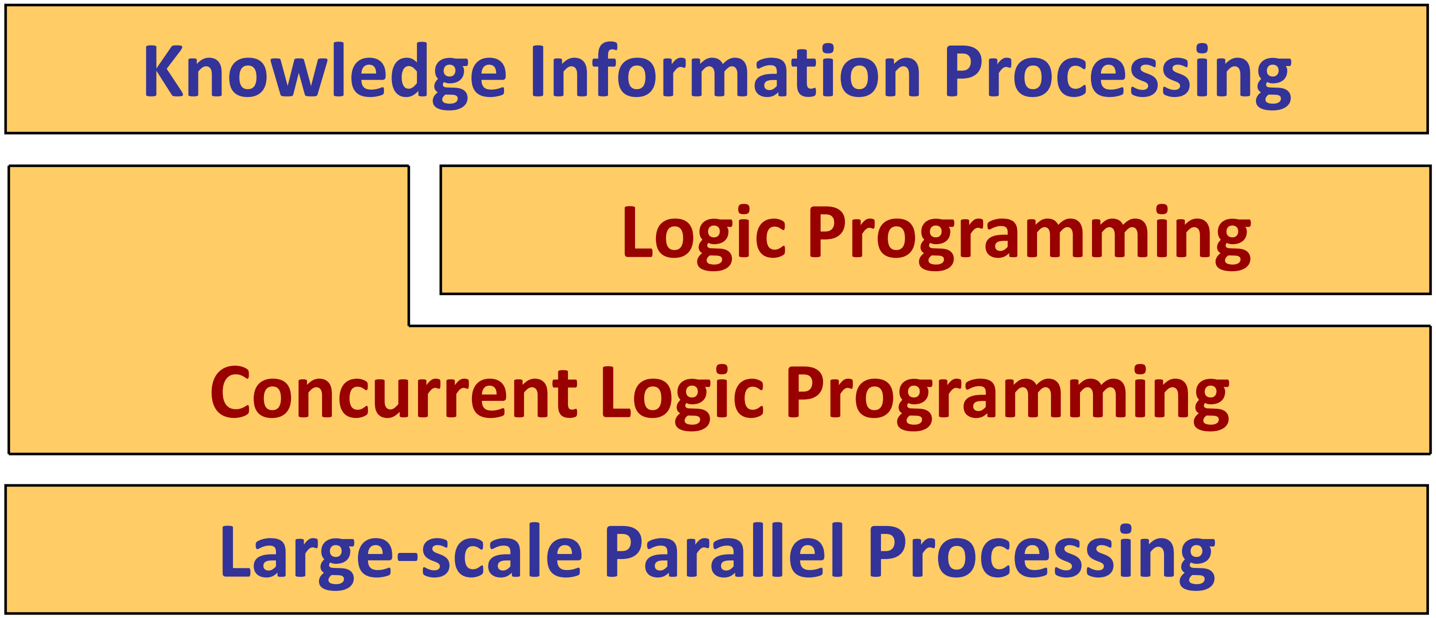
Kernel Languages

- ***Kernel Languages* as the core of the project**
  - **KL0**: Sequential kernel language for startup
  - **KL1**: Parallel kernel language for systems and apps
  - (KL2: Knowledge representation language)

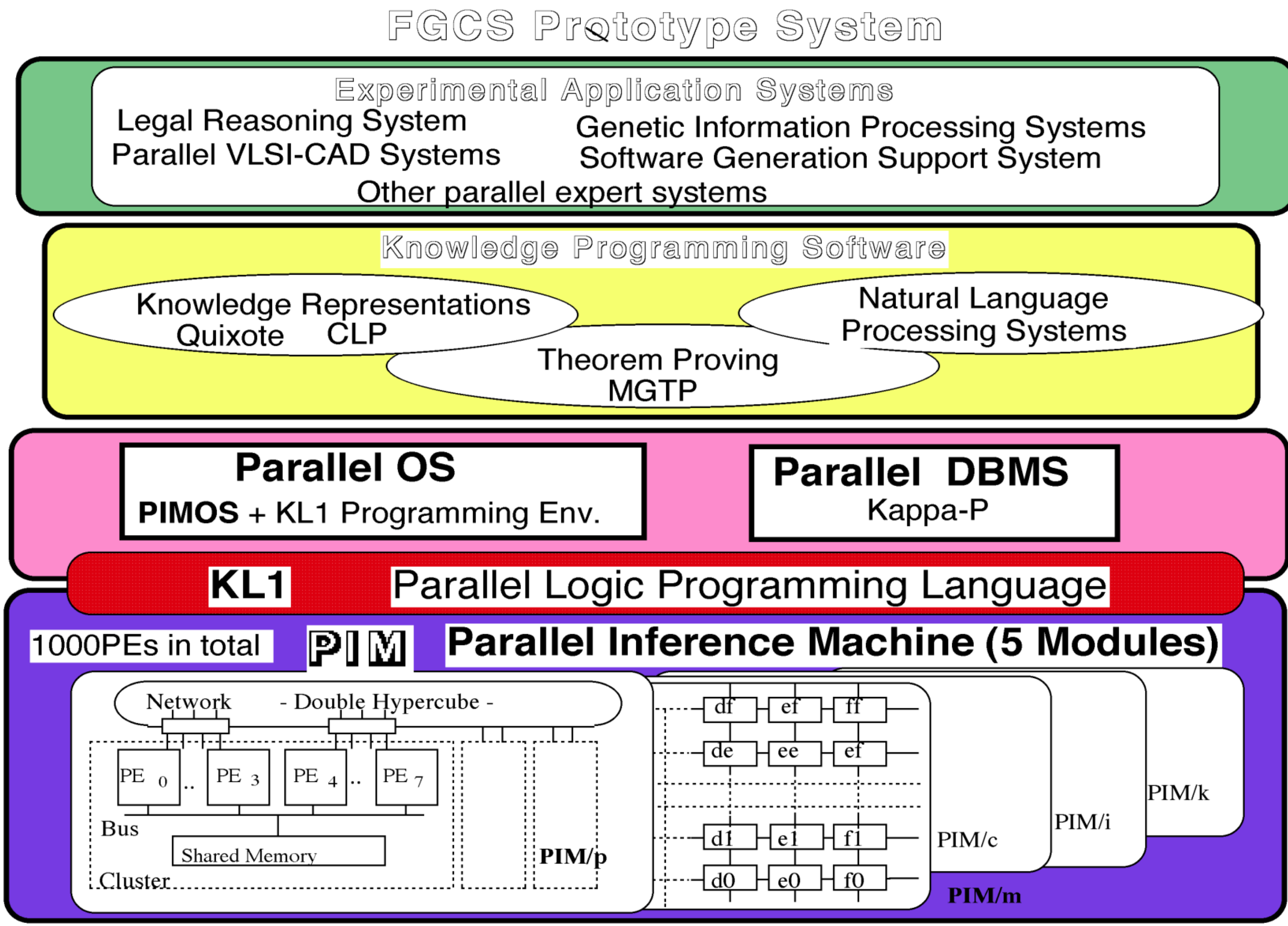


Outcome of the Project as of 1993

- Basic structure:

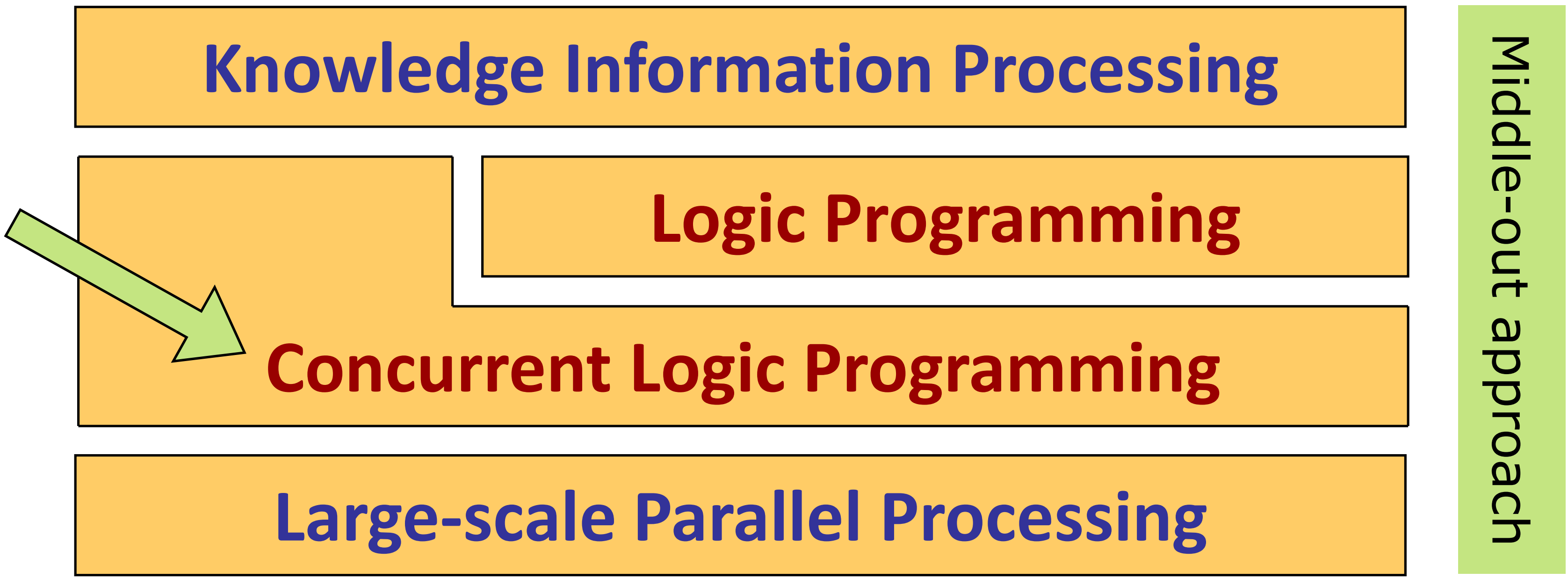


Outcome of the Project as of 1993



Outcome of the Project as of 1993

- **Some key outcomes:**
  - Kernel Language (GHC/KL1)
  - Parallel OS (PIMOS) totally written in KL1
  - Parallel Inference Machine (PIM with 512 & 256PEs)
  - Model Generation Theorem Prover (MGTP)



Hardware built and heavily used

Kazuhiro Fuchi

PSI-I (Dec. 25, 1983)
35KLIPS for KL0
80MB, 100 copies
SIMPOS in ESP
+ many apps

D.H.D. Warren

PSI-II (Dec. 1986)
330 (400) KLIPS for KL0
with 6.67MHz, WAM, 300 copies

Hardware built and heavily used

Shunichi Uchida

Multi-PSI (Mar. 1988)
64 PSI-II's in 2D mesh
5MLIPS for KL1, 6copies
PIMOS (standalone,
multi-user OS in KL1,
44KLOC in 0.5years)

PIM/m (1992)
256 PEs in 2D mesh (256 PEs)
200 MLIPS, PIMOS (200KLOC) in KL1

Kernel Language [CACM March 1993 issue]

Overview of the FGCS Project
Kernel Language
Constraint-Based Concurrency
Demos
Perspectives

KL1 Design Task Group (April 1983-)

- Led by Koichi Furukawa
- Requirements included:
  - General-Purpose
  - Parallel algorithms
  - Operating systems
  - Meta-programming

Keith Clark Ehud Shapiro

- Invited Keith Clark, Steve Gregory (PARLOG) and Ehud Shapiro (Concurrent Prolog) in October 1983
- Had many meetings and exciting discussions

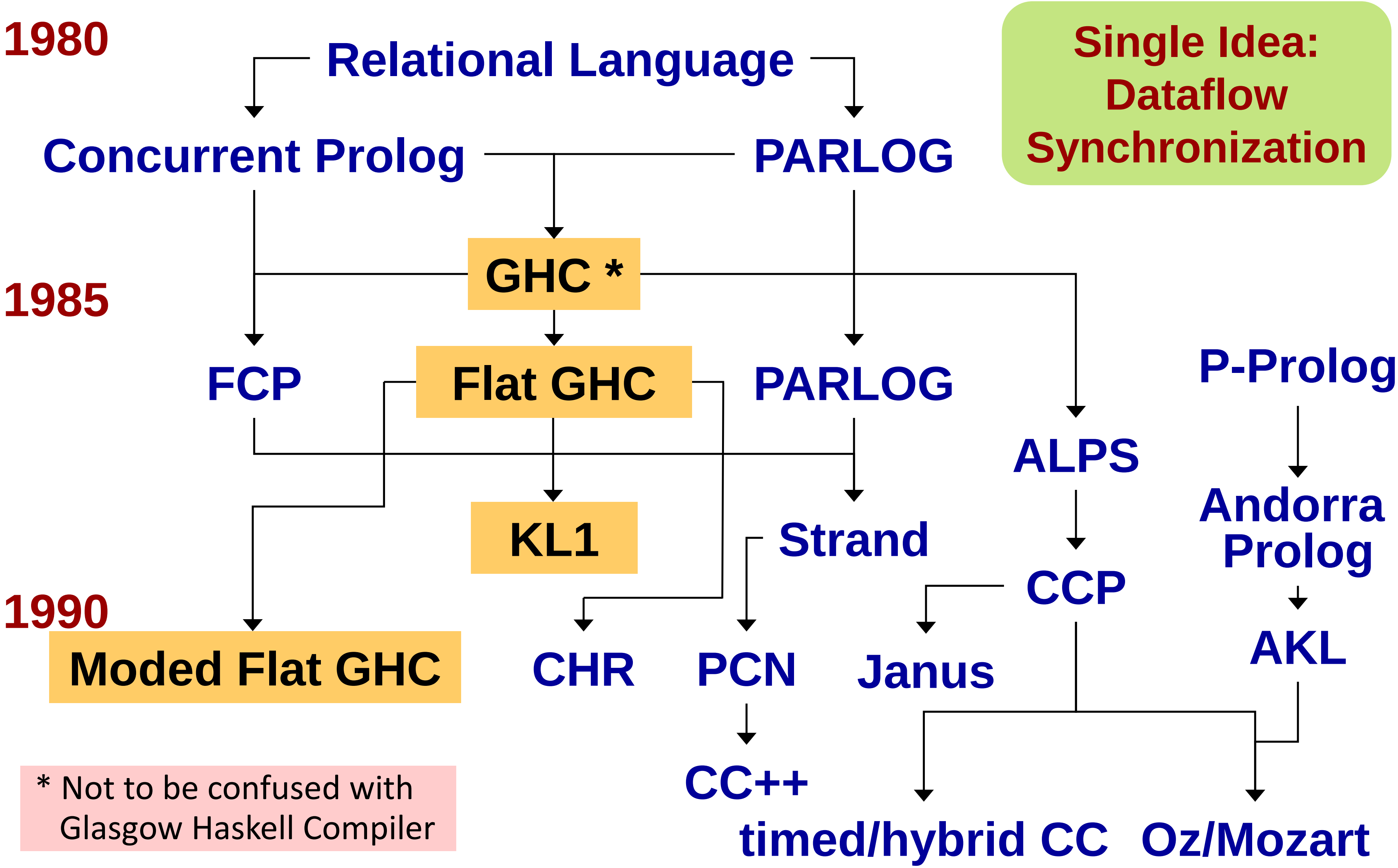
Concurrent Logic Languages

- Syntax: **Guarded clauses** (cf. Guarded Commands)
- Semantics:
  - **Dataflow synchronization** (cf. sequentiality)
  - **Commitment** (don't-care nondeterminism)

➔ *Communicating Parallel Processes*

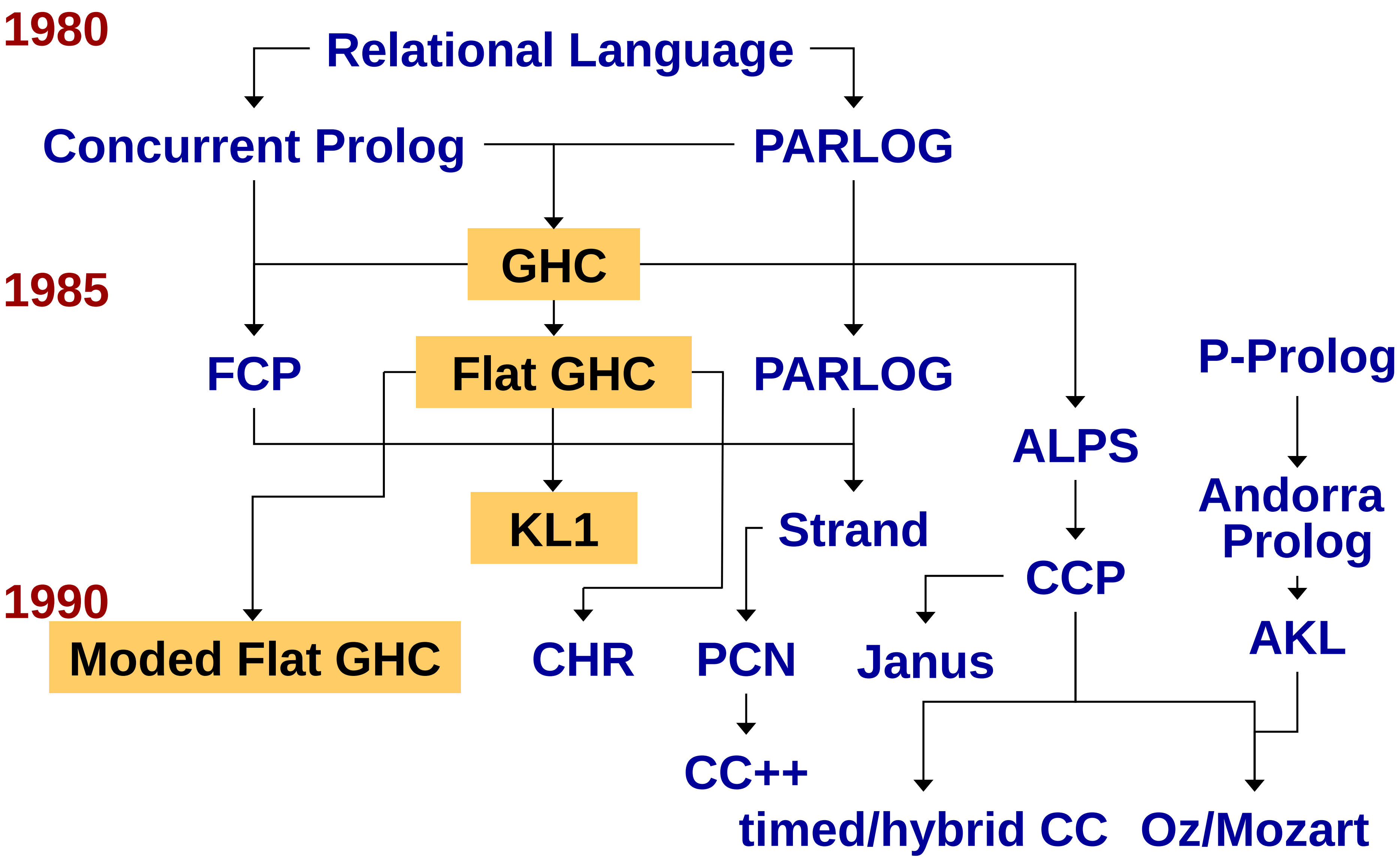
Processes ↔ **(AND-parallel) Goals**
(actors, agents, objects, …)
Channels ↔ **Logical variables used as streams**

Early History of Constraint-Based Concurrency



* Not to be confused with Glasgow Haskell Compiler

Early History of Constraint-Based Concurrency

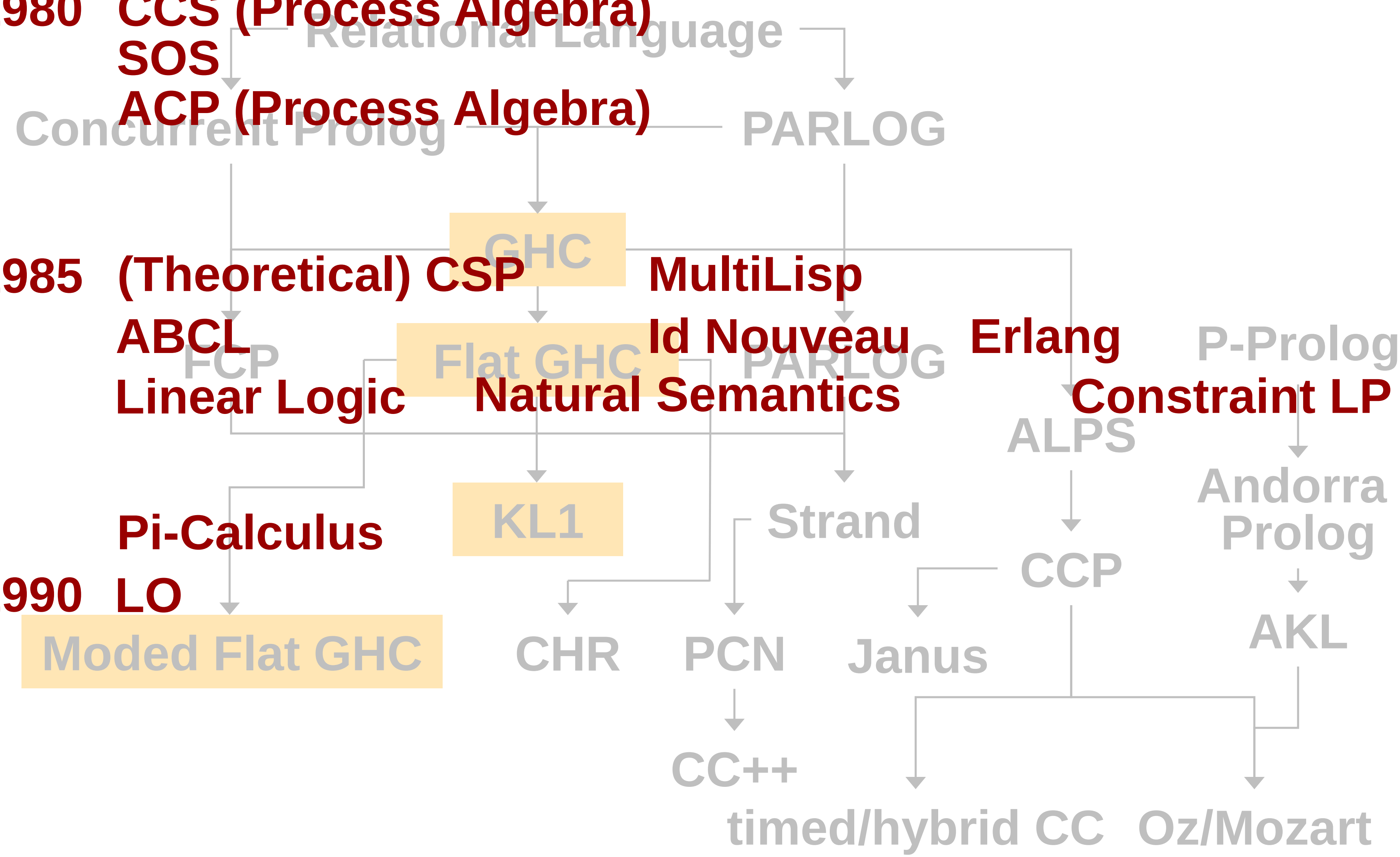


Early History of Constraint-Based Concurrency

1980 CCS (Process Algebra)
SOS
ACP (Process Algebra)

1985 (Theoretical) CSP — MultiLisp
ABCL — Id Nouveau — Erlang
Linear Logic — Natural Semantics — Constraint LP
Pi-Calculus

1990 LO



Some useful discussions (1983)

- Norihisa Suzuki (Univ. Tokyo, working on Smalltalk):
  "Building the entire system without resort to side effects is the FGCS project's right way to go."

- Shunichi Uchida (ICOT):
  "The computational model of KL1 should not assume any particular granularity of underlying parallel hardware."
  (= Kernel Language should embrace as fine-grained concurrency as possible.)

Some useful discussions (1983)

- Ehud Shapiro (Weizmann Inst., in response to the first requirement specification of KL1)
  "Too many good features."

  ➔ The TG's basic research question then became:

> *"What's the minimum set of language constructs that turn Logic Programming into an expressive concurrent language?"* (Occam's Razor, cf. CSP/Occam)

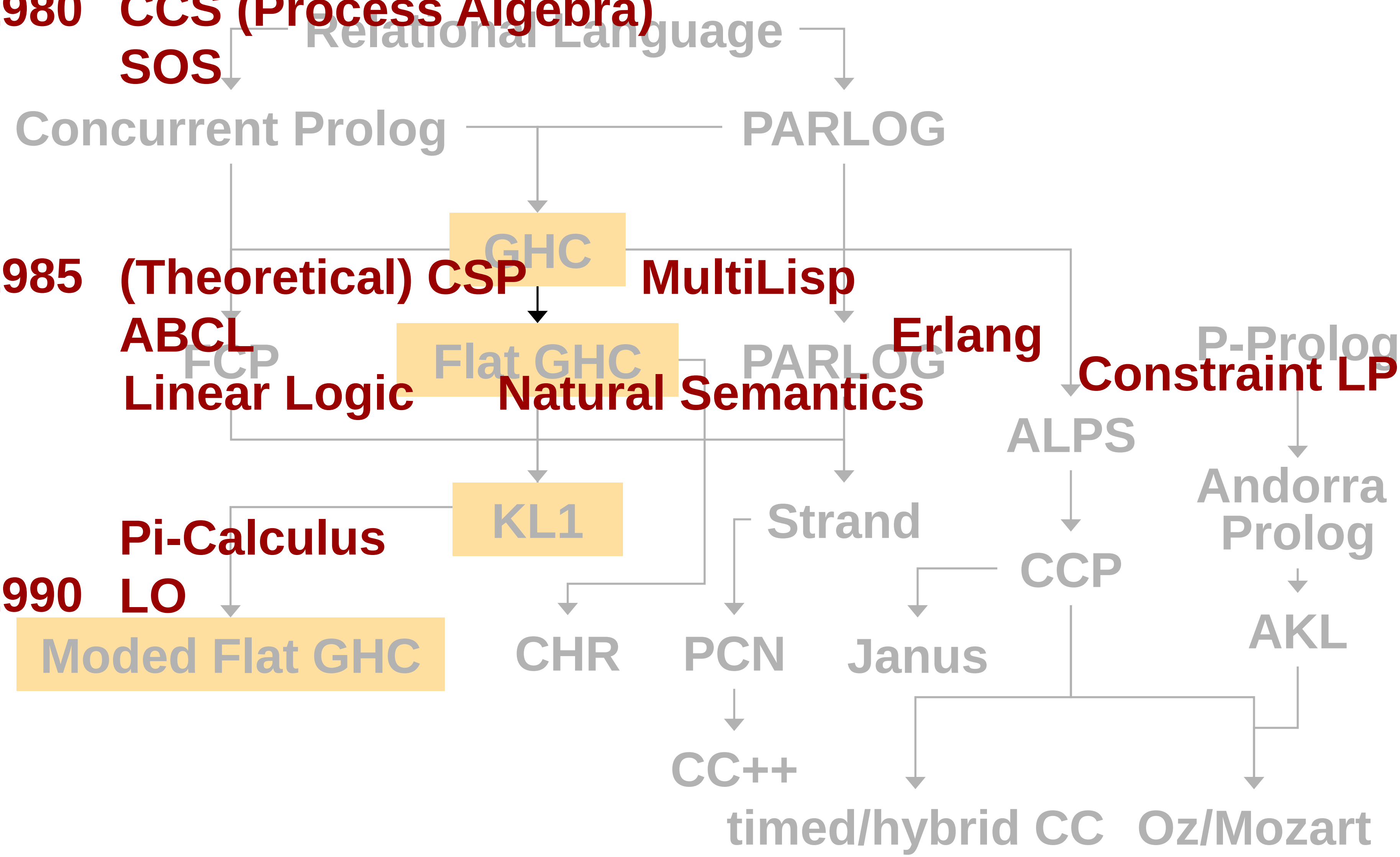
- Concurrent Prolog chosen in February 1984 as the "refined working hypothesis" of KL1
  - with some uneasiness

Early History of Constraint-Based Concurrency

1980 CCS (Process Algebra)
SOS

1985 (Theoretical) CSP MultiLisp
ABCL Erlang
Linear Logic Natural Semantics Constraint LP

Pi-Calculus
1990 LO



Addressing the research question

Three approaches to dataflow synchronization

- **PARLOG**: I/O modes (polarities) to goal arguments
- **Concurrent Prolog**: Read-only annotation given to logical variables (cf. capabilities), which allowed
  - *Incomplete messages (messages with reply boxes)*
  - *Channel mobility*

  Highly expressive (cf. π-calculus in late 1980's),
  but efficient distributed implementation seemed rather difficult (after intensive implementation effort).

Addressing the research question

- **Guarded Horn Clauses (GHC)** (Dec. 1984) [LNCS 221]:
  Guards are suspended if they attempt to export bindings
  - Incomplete messages and channel mobility retained with no additional syntactic constructs
- Quickly adopted as the new working hypothesis of KL1 (early 1985)
  - welcomed by the hardware group
  - Lesson: *simplicity was the key to the consensus*.

GHC put in another way:

- Parallel execution of logic programs requires the management of *multiple binding environments.*

  Q: *"What's the minimal mechanism that enables parallel execution of logic programs in a single binding environment?"*

  A: "Just suspend computation which would otherwise publish bindings!"

  Q: "Then, when can a process publish bindings?"

  A: "When it's the only one to do so."

Guarded Horn Clauses [LNCS 221]

- Working prototype implementation (in DEC-10 Prolog) in 1.5 days
  - Adapted from Concurrent Prolog compiler on top of Prolog [SLP 1985]

    www.ueda.info.waseda.ac.jp/~ueda/software/ghcsystem-swi.tgz
  - Another prototype implementation in CAML by Gerard Huet (INRIA) in a few days (1988)
- Quickly subsetted to Flat GHC (mid 1985)
- Encoding of search (*findall*) devised [ICLP 1986]

Guarded Horn Clauses (end of 1984)

DESIGN PRINCIPLES OF GHC

1. Parallelism

- It must be a parallel language 'by nature', not a sequential language augmented with constructs for parallelism,
  - — to have a clearer semantics, and
  - — to disallow inessential sequentiality to creep in.
- Introduction of sequentiality is considered as an optimization to meet the current computer architectures.
- We have to allow even possibly useless computation.

2. Generality

- It must be a general-purpose language which can express important concepts in parallel programming.
- It must be general also in that no specific implementation scheme is assumed a priori.

3. Simplicity

- It must be a simple language because of the shortage of our experience both in the theoretical and the practical aspects of parallel programming (languages).

4. Efficiency

- It must be an efficient language which allows fast execution of simple programs at least under the current computer architectures.
  - — cf. generality (2.)
- Sequential implementation is more than a prototype.
- Efficiency may interfere with generality and simplicity, but a general language could be subsetted for more efficient execution of a specific class of programs.

GHC syntax

(program)	P ::= set of R's
(program clause)	R ::= A :- | B
(body)	B ::= multiset of G's
(goal)	G ::= $T_1 = T_2$ | A
(non-unif. atom)	A ::= $p(T_1, \ldots, T_n)$, $p \neq$ '='
(term)	T ::= (as in first-order logic)
(goal clause)	Q ::= :- B

GHC syntax

(program)	P ::= set of R's
(program clause)	R ::= A :- | B
(body)	B ::= multiset of G's
(goal)	G ::= $T_1 = T_2$ | A
(non-unif. atom)	A ::= $p(T_1, \ldots, T_n)$
(term)	T ::= (as in first-order logic)
(goal clause)	Q ::= :- B

tell

rewrite rule with *ask*, choice, reduction & hiding

parallel composition

From GHC to KL1

- GHC = *concurrent* language model
- KL1 = *parallel* language with
  - mapping
  - protection

- Lesson: *separation of concerns*
  - *concurrency vs. parallelism*
  - *concurrency vs. search* (cf. multiparadigm langs)
  - *reduction vs. communication*
    (atomic vs. eventual tell controversy, cf. π-calculus)

Constraint-Based Concurrency

— what we developed in mid 1980s, in retrospect

Overview of the FGCS Project
Kernel Language
Constraint-Based Concurrency
Demos
Perspectives

Marriage of CLP and CLP

- **Concurrent Constraint Programming** (late 1980's)
  - *Formalization of* Concurrent Logic Programming *inspired by* Constraint Logic Programming
  - Logical view of communication (**Ask** / **Tell**)
  - Abstraction and generalization of data domains

→ **Constraint Based Concurrency** based on
  - ***single-assignment*** **(write-once) channels** and
  - **constructors**

(cf. Name-Based Concurrency, i.e., CCS, CSP, π, etc.)

Single-Assignment Channels

- Also known as *logical variables*
- Can be written at most once
  - by ***tell***ing a constraint (= partial information) on the value of the channel (*unification*)
    - e.g., tell S=[read(X)|S']
- Reading is non-destructive
  - by ***ask***ing if a certain constraint is entailed (*term matching*)
    - e.g., ask $\exists$A $\exists$S'(S=[A|S'])
  - covers both *input* and *match* in the π-calculus

Single-Assignment Channels

- The set of all published constraints (***tell***s) forms a *constraint store*.
- Since reading is non-destructive, constraint store is monotonic.
  - Still, it's amenable to garbage collection because of its highly local nature.

Constraint-Based Communication

- Asynchronous
  - ***tell*** is an independent process (as in the asynchronous π-calculus)
- Polyadic ("many-place")
  - constructors provide built-in structuring and encoding mechanisms
  - essential in the single-assignment setting
- Mobile
- Non-strict

Constraint-Based Communication

- Asynchronous
- Polyadic
- Mobile – channel mobility in the sense of the π-calculus
  - Channels
    - can be passed using another channel
    - can be fused with another channel
    - are first-class (processes aren't)
  - available since 1983 (Concurrent Prolog)
- Non-strict

Constraint-Based Communication

- Asynchronous
- Polyadic
- Mobile
- Non-strict
  - "Constraint-based" means *computing with partial information*
  - Yielded many programming idioms, including
    - (streams of)* streams
    - difference lists
    - messages with reply boxes

Channels in CBC Are Local Names

- Fallacy: constraint store is global, shared, single-assignment memory
- Channels are created as *fresh local names* that cannot be forged by the third party and can be transmitted only by using an existing channel
  - e.g., p([create(S)|X’]) :- | server(S), p(X’).
- Thus, constraint store allow us to model *secure, mobile, peer-to-peer* communication network.

Demos

Overview of the FGCS Project
Kernel Language
Constraint-Based Concurrency
Demos
Perspectives

KLIC (KL1-to-C translator)

- Developed in the Two-Year Follow-up project (1993-1994, 2.8BJPY)
- Still runs (made to run!) in parallel on shared-memory Linux machines with many cores
  - Single-core execution is 10x faster than 10 core execution of 20 years ago.
- Lesson: *Old software is lightweight and fast. Why not keep it alive?*
  - ... though C applications require maintenance.

MGTP/G

- Model-Generation Theorem Prover
  - Won IJCAI’93 award by solving open problems in group theory
  - Apps: Disjunctive Databases, Abductive Inference, Legal Inference (HELIC-II), Constraint Satisfaction, …
- Handles clauses of the form

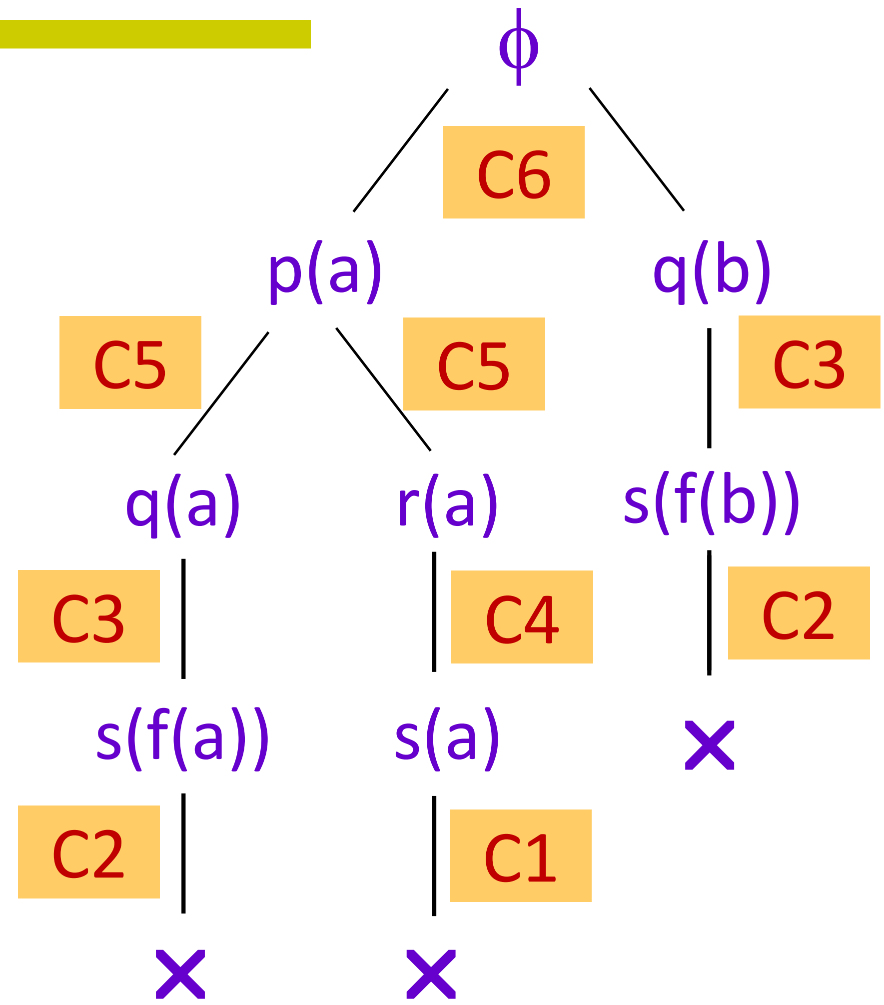
  C1: p(X), s(X) $\rightarrow$ false.
  C2: q(X), s(Y) $\rightarrow$ false.
  C3: q(X) $\rightarrow$ s(f(X)).
  C4: r(X) $\rightarrow$ s(X).
  C5: p(X) $\rightarrow$ q(X); r(X).
  C6: true $\rightarrow$ q(a); q(b).

MGTP/G

- Handles of the form

  C1: p(X), s(X) → false.
  C2: q(X), s(Y) → false.
  C3: q(X) → s(f(X)).
  C4: r(X) → s(X).
  C5: p(X) → q(X); r(X).
  C6: true → p(a); q(b).

- Compiled into KL1, translating OR-parallelism into controlled AND-paralleism



Perspectives

Overview of the FGCS Project
Kernel Language
Constraint-Based Concurrency
Demos
Perspectives

Offspring of Concurrent LP

- **Concurrent Constraint Programming** (late 1980's)
  - Inspired by Constraint Logic Programming
  - Logical view of communication (**Ask** / **Tell**)
  - Generalization of data domains (esp. multisets)
- **CHR (Constraint Handling Rules)** (early 1990's)
  - Allows multisets of goals in rule heads
  - An expressive multiset rewriting language
  - Many applications (esp. constraint solvers)
- **Timed / Hybrid CCP** (early-mid 1990's)
  - Introduced time, defaults, and continuous change
  - High-level language for timed and hybrid systems

Offspring of Concurrent LP

- **Languages for High-Performance Parallel Computing and Grids** (early 1990's and on)
  - PCN, CC++, HPC++, swift-lang

from Ian Foster @ ANL

> Dear Ueda-san:
> The wonders of Google Scholar citation alerts led me to your recent paper on FGCS, which I enjoyed reading.
> …
> While PCN and CC++ are long gone, we continue to work with Swift (swift-lang.org), which is really CLP in another guise.
> My best wishes from Chicago.

- **X10** (mid 2000's)
  - IBM's solution to HPC languages

Offspring of Concurrent LP

- **LMNtal** (2002) http://www.ueda.info.waseda.ac.jp/lmntal/
  - Integration of processes and data, single name category
    - (FP) functions vs. constructors
    - (LP) predicates vs. functions

  ➔ Multiset (many-to-many) rewriting (a la CHR) with zero-assignment logical variables
  (= graph rewriting language)

  - Allows encoding of various calculi including strong λ
  - State-space search is now back with LTL model checking

Lessons learned [ALP Newsletter 2006]

(iv) different concerns should be separated to understand things analytically; only after that they could be integrated.

(v) A good way to understand and examine a language definition is to implement it; it forces us to consider *every detail* of the language.

(vii) The small-step semantics of a language construct does not necessarily express the real atomic operations of the construct.

(xx) Logic programming today embraces diverse interesting technologies beyond computation logic as well as those within computational logic.

Further Readings

- **The Fifth Generation Project: Personal Perspectives**

  - CACM, **36**(3), 1993 (D.H.D. Warren & E. Shapiro, eds.)

  - Kazuhiro Fuchi (ICOT Director) pp.49-54
  - Robert Kowalski pp.54-60
  - Koichi Furukawa pp.60-65
  - Kazunori Ueda pp.65-76
  - Ken Kahn pp.77-82
  - Takashi Chikayama (principal implementor of KL1/PIMOS) pp.82-90
  - Evan Tick pp.90-100

Further Readings

- **Concurrent Logic/Constraint Programming: The Next 10 Years**

  - In *The Logic Programming Paradigm: A 25-Year Perspective*, Springer, 1999, pp. 53-71.

- **Logic Programming and Concurrency: a Personal Perspective**

  - *The ALP NewsLetter*, **19**(2), 2006 (6 pages).

- **Fifth-Generation Computer Systems Museum**

  - AITEC-ICOT Archives DVD, 2005

Thank you for your attention.

Questions and off-line discussions welcome.

Computing paradigms must change . . .

20th century	21st century
von Neumann architecture + sequential computation	multi-core / clusters / Grid / distributed / embedded / molecular / ...
◆ Turing Machines (computability) ◆ RAM model (complexity) ◆ λ-calculus (programming languages) ◆ Floating point arithmetic (numerical analysis)	?

Computing paradigms must ch

Concurrency Everywhere!

20th century	21st century
von Neumann architecture + sequential computation	multi-core / clusters / Grid / distributed / embedded / molecular / ...
◆ Turing Machines (computability) ◆ RAM model (complexity) ◆ λ-calculus (programming languages) ◆ Floating point arithmetic (numerical analysis)	? What to teach at Universities?

Concurrent Logic/Constraint Programming: The Next 10 Years

Kazunori Ueda

Waseda University

Grand Challenges

- A "λ-calculus" in concurrency field

  cf. *X*-calculus (calculus of *X*)

  X: π, action, join, gamma, ambient, ...
- Common platform for non-conventional computing (parallel, distributed, embedded, real-time, mobile, ...)
- Type systems (in the broadest sense) and frameworks of analysis for both logical and physical properties

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

LP vs. Concurrent LP

- Concurrent LP = LP + 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

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

I/O Modes: Motivations

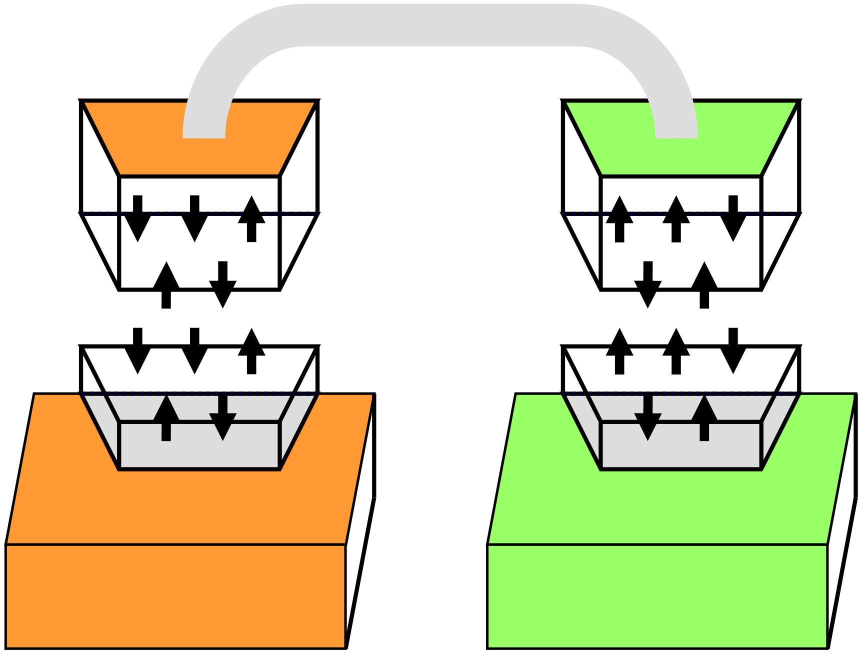
- Our experience with concurrent logic languages (Flat GHC) shows that logical variables are used mostly as *cooperative* communication channels with statically established protocols (point-to-point, multicasting)
- Non-cooperative use may cause collapse of the constraint store
  - e.g., $X=1 \wedge X=2 \wedge 1\neq 2$ entails anything!

The Mode System of Moded Flat GHC

- Assigns *polarity (+/–) structures* to the arguments of processes so that the write capability of each part of data structures is held by exactly one process
- Unlike standard types in that modes are resource-sensitive
- Moding rules are given in terms of mode constraints (cf. inference rules)
- Can be solved (mostly) as unification over mode graphs (feature graphs with cycles)

An Electric Device Metaphor

- Signal cables may have various structures (arrays of wires and pins), but
  - the two ends of a cable, viewed from outside, should have opposite polarity structures, and
  - a plug and a socket should have opposite polarity structures when viewed from outside.



goal = device
variable = cable

Moding: Implications and Experiences

- A process can pass a (variable containing) write capability to somebody else, but cannot duplicate or discard it.
- Two write capabilities cannot be compared
- Read capabilities can be copied, discarded and compared
  - cf. Linearity system
- Extremely useful for debugging – pinpointing errors and automated correction (!)
- Encourages resource-conscious programming

Moding: Implications and Experiences

- Encourages resource-conscious programming by giving weaker mode constraints to variables with exactly two occurrences
  - A singleton variable constrains the mode of its position to fully input or fully output.
  - A variable with three or more occurrences constrain the modes of more positions.
- Weaker constraints lead to more generic (= more polymorphic) programs