FGCS 计划的基本研究活动

刘海燕　张　静　编译　王献昌　审校

摘　要　FGCS 计划的目标是开发知识信息处理的并行计算机。ICOT 预测逻辑程序设计技术可以解决知识处理的并行化问题，并强调逻辑程序设计两个不同的方面：一是建立新的信息技术，二是追求基本的 AI 和软件工程研究。研究成果表明，ICOT 的预测是正确的，所采取的方法合理。

1. 引言

FGCS 计划追求两个研究目标：知识信息处理和并行处理，为此，ICOT 把逻辑程序设计作为关键技术。在计划的开始，以 Prolog 为工具来推动计划的整个研究，其中有三项成果引人注目。第一是为 ESP 开发了工作站，并完全用 ESP 实现了工作站的 OS。第二是对元程序设计部分评估的研究。第三是开发了约束逻辑程序设计语言：CIL 和 CAL。

2. 并发逻辑程序设计(CcLP)

并发逻辑程序设计研究的两个重要课题是：并发逻辑程序设计语言 Flat GHC(FGHC)的设计原理和 FGHC 的搜索风范。

2.1　FGHC 的设计原理

FGHC 的一个重要特征是它对 Prolog 只有一个语法扩充，称之"约定操作符(commitment operator)"并用竖线"|"表示。它把子句分成两部分，卫士(guard)部分("|"的左边)和体部分("|"的右边)。卫士有两个重要作用：一是指明选择子句进行后继计算的条件；二是指明同步条件。FGHC 的同步规则是"数据流同步"，意指计算被挂起直到足够的数据到达。在 FGHC 中，卫士一直被挂起直到调用者被充分实例化以判断卫士条件为止。比如，考虑一个售票机的工作：收到钱后，它必须等到用户按下一个目的键。这个等待可描述为一个子句："if the user pushed the 160-yen button，then issue a 160-yen ticket"。

重要一点是，数据流同步可通过一简单的规则实现。当一个目标正被执行且正调用相应的 FGHC 子句时，该规则被一致化。头一致化(head unification)信息流必须是单向的，即从调用者到被调用者。例如：用两个子句定义谓词：桌前服务。一个在白天，当期望有更多的顾客时；一个是下班后，当不期望有顾客时。子句定义如下：

```
serve([First|Rest]):-〈extra-condition〉|
        do_service(First),serve(Rest).
serve([]):-true|true.
```

还有一个进程 queue，它为服务组织一个等待队列。顶层目标是：?— queue(Xs),serve(Xs). 其中"?—"是终端用户提示符。该目标的执行产生两个进程：queue 和 serve。它们共享变量 Xs，Xs 作为管道把数据从一个进程传给另一个进程。现假定 queue 进程对 Xs 实例化，serve 进程读取 Xs 值，即 queue 充当 Xs 的生产者，serve 充当消费者。queue 进程为一个服务对象表[(first-servee)，(second-servee)，…]，或为一个空表[]实例化 Xs，实例化前 Xs 无定义。

假设 Xs 无定义，由于未实例化的 Xs 使方程 Xs＝[First|Rest]和 Xs＝[]无解，因此目标 serve(Xs)的调用使头一致化挂起。但这样的实例化违反了单向一致化规则。注意：serve 头的项[First|Rest]意味着这个子句期望得到一个非空表作参数值，同样，[]则期望得到一个空表。显而易见，信息流的单向化实现了数据流同步。

这一设计原理很重要，有两个方面：一是 FGHC 语言为表达并发提供了一个自然的工具；二是简单的同步机制可实现高效的并行化。

2.2　FGHC 的搜索风范

约定式选择的特征使 FGHC 缺乏自动搜索能力，这恰是 Prolog 的特征，即 Prolog 可通过自动回溯进行搜索。约定式选择唯一决定了成功计算目标的子句，那么就无法搜索除被选分枝外的其它分枝。搜索能力与逻辑程序设计计算过程的完备性有关，因此缺乏搜索能力是很严重的。由于某些原因，使得采用 OR-并行搜索或类 Prolog 语言作为并行计算的内核语言都行不通。ICOT 试图通过程序设计技术同时保持程序设计语言尽可能简单来恢复搜索能力，他们成功地提出了几种程序设计方法来计算一

个问题的所有解，从而实现了逻辑程序设计的完备性。如①基于继续(Continuation_based)的方法。②分层流式(Layered stream)方法。③查询编译(Query compilation)方法。

本文论述二者互补的方法①和③。基于继续的方法适于颇有算法性(rather-algorithmic)的问题的高效处理，如计算使用 append 把一给定表分解成两个子表的所有分解方式。此方法使用 FGHC 的 AND-并行模仿 OR-并行 Prolog 的计算，Prolog 中的 AND-串行计算被翻译为继续(continuation)处理，它记住继续栈(保存作为 FGHC 目标的参数之一)中的继续点。计算的中间结果通过继续栈从前一个目标传到下一个目标。此方法要求把 Prolog 程序翻译成 FGHC 之前对输入/输出方式进行分析。由于每个谓词有太多的输入/输出方式，故该方法不适合数据库的应用。

查询编译方法解决了这个问题，是在 KL1 开发一个自底向上定理证明器时首次提出的。该方法用一个编译后的子句表示每个询问成分，用一个由目标传递的数据结构表示数据库，努力寻求一个询问成分子句来匹配表示一个记录的目标，并把数据库中所有用于潜在的可应用记录的进程递归化，从而取代了寻找一个记录(用子句表示)去匹配一个由目标表示的给定查询模式。由于每个记录是一个基本项(ground term)，所以调用者中无变量。当搜索到询问成分子句时，若发现一子句匹配当前正处理的记录，则发生变量实例化。注意：信息流向是从调用者(数据库)到被调用者(一个询问成分)，信息流的反向避免了查询处理的死锁。一个重要技巧是：每次调用一个查询子句，就为询问成分中每个变量产生一个新值，对每个 OR-并行计算分枝创造一个新环境，用 KL1 变量表示数据库查询中的目标一级变量，避免了对整个数据库和对 I/O 模式查询的不同编译。

3. 约束逻辑程序设计(CsLP)

FGCS 计划初，联系自然语言处理，开始了 CsLP 的研究，Mukai 为实现一个环境语义的可计算模型，开发了 CIL (Complex Indeterminates Language)，但它的求解约束的方式是被动的，没有诸如求解约束传播或联立方程之类的约束求解的主动计算。后来，研究了包括主动约束求解的 CsLP，推出了 CAL，它能处理表达约束的非线性方程。对这一研究做了三件有关的事：1)为线性代数定理证明器开发了 METIS 的项重写系统；2)为求解非线性方程采纳了计算 Gröbner 基(basis)的 Buchberger 算法；3) Jaffar 和 Lassez 提出了 CsLP(X)理论，以构成 CsLP 语言的一个框架。

对一般符号处理领域中的 CsLP 研究，Tsuda 开发了 cu-Prolog 语言，其约束求解采用了程序展开/折叠(unfold/fold)变换技术。每次执行变换时，程序被修改为语法上约束较少的程序。展开/折叠作为基本操作，类似于 CAL 的项重写基本操作，二者都是通过重写程序来获得规范形式。cu-Prolog 思想吸收了 Hasida 的依赖传播和动态程序设计的成果。

CsLP 极大丰富了 Prolog 的表达能力，它通过扩充 Prolog 定义域以覆盖大部分 AI 问题，为应用提供了很有前途的程序设计环境。FGCS 计划面临的一个主要问题是怎样把 CsLP 与并发逻辑程序设计结合起来以便同时获得较强的表达能力和高效性。但这种结合并非易事，因为①CsLP 主要集中在高效执行每个约束求解的控制方案上；②CsLP 包括一个搜索风范，它需要象自动回溯那样合适的支持机制。使用数据流控制法的并发逻辑程序设计方案在某种程度上可有效处理第①个问题；但第②个问题迄今尚未解决，不过存在两种可能的办法：开发一个 OR-并行搜索机制或采用新编码技术以有效地引入搜索风范。

4. 高级软件工程

软件工程的目的是支持开发高质量、易维护的软件和提高软件生产率。逻辑程序设计在软件工程的诸多方面具有很大潜力。

4.1 元程序设计和部分求值

ICOT 以元程序设计技术为工具，开发了基于知识的系统，把完整约束看作知识库必须满足的元规则。元解释是在每个目标级程序之上，难免低效。Takeuchi 和 Furukawa 通过对元程序使用部分求值的优化技术对这个问题有所突破。首先，在给定规则(带确定性因子)的元解释器后，获得了带不确定性计算的专家系统的一个编译后的程序；其次开发了一个自底向上的语法分析器的元解释器并且在给定解释器和一组语法规则后，获得一高效编译后的程序。

继把部分求值技术应用于元程序设计之后，Fujita 和 Furukawa 成功地研制出一个简单的自适用的部分求值器(self-applicable partial evaluator)，它是一个元解释器，类似于下面的 Prolog 解释器：

```
solve(true).
solve((A,B)):-solve(A),solve(B).
solve(A):-clause(A,B),solve(B).
```

其中假定对每个程序子句 H:-B，一个单位子句 clause(H,B)是可断言的。一个目标 solve(G)模拟源目标 G 的一次立即执行并获得相同的结果。

Prolog 自解释器的简单定义(solve)暗示了下列的部分求解器 psolve.

```
psolve(true,true).
psolve((A,B),(RA,RB)):-
    psolve(A,RA),psolve(B,RB).
psolve(A,B):-
    clause(A,B),psolve(B,R).
psolve(A,A):-'$suspend'(A).
```

部分求解器 pslove(G,R)部分求解一个给定目标 G 并返回结果 R，R 是给定目标 G 的剩余目标(residual goal(s))。若 G 完全求解，则剩余目标为 true，否则它是子目标的合取，每一个合取项是目标 R_i，在 '$suspend'($R_i$)处被挂起以求解。附加谓词 '$suspend'(P)是由用户为每个目标模式 P 定义的。注意 psolve 和 solve 的关系是：solve(G):-psolve(G,R),solve(R)，即如果目标 G 部分求解且剩余目标 R 也成功完全求解，则目标 G 成功。因此 G 的完全求解分成两个任务：G 的部分求解和剩余目标 R 的完全求解。

在短期内 ICOT 在 Prolog 框架中开发了可自适用的部分求值器，证明了逻辑程序设计语言有超越函数式语言的优势，特别在其基于一致化的束定(binding)模式方面。

4.2 展开/折叠程序变换

程序变换为软件开发提供了强有力的方法论，尤其是从形式说明或从低效的说明性程序导出有效的程序。Prolog 标准的自左至右控制规则是用说明形式写的程序往往低效的所在之处，基于生成(generate)和测试(test)范型的程序则是典型的例子。Seki和 Furukawa 为这些程序开发了展开/折叠的程序变换方法。谓词 gen-test(L)定义如下：

```
gen-test(L):-gen(L),test(L).
```

其中 L 是变量，表示一个表，gen(L)是表 L 的生成器，test(L)是表 L 的测试器。假定 gen 和 test 是递增的，定义如下：

```
gen([])
gen([X|L]):-gen_element(X),gen(L).
test([]).
test([X|L]):-test_element(X),test(L).
```

那么可通过应用展开/折叠变换将 gen 和 test 融合：

```
gen_test([X|L]):-gen([X|L]),test([X|L]).
      unfold at gen and test
gen_test([X|L]):-gen_element(X),gen(L),
    test_element(X),test(L).
      fold by gen_test
gen_test([X|L]):-gen_element(X),
    test_element(X),gen_test(L).
```

如果测试器(tester)不是递增的，上面的展开/折叠就行不通。若测试表中所有元素互不相同，test 谓词定义如下：

```
test([]).
test([X|L]):-non_member(X,L),test(L).
non_member(_,[]).
non_member(X,[Y|L]):-
     dif(X,Y),non_member(X,L).
```

dif(X,Y)判断 X 不等于 Y，注意 test 谓词不是递增的，因为对第一个元素 X 的测试要求全表信息。解决的办法是通过增加一个累加器作 test 谓词的附加参数而得到一等价谓词以代替 test 谓词。定义如下：

```
test'([],_).
test'([X|L],Acc):-
     non_member(X,Acc),test'(L,[X|Acc]).
```

test 和 test' 之间的关系由下面定理给出：

定理 test(L)≡test'(L,[])

原来的 gen-test 程序变为：gen_test(L):-gen(L),test'(L,[]). 需要引入下列新谓词来执行展开/折叠变换：gen_test'(L,Acc):-gen(L),test'(L,Acc). 通过应用与前面类似的变换过程，得到下面融合的递归的 gen-test' 程序：

```
gen_test'([],_).
gen_test'([X|L],Acc):-gen_element(X),
     non_member(X,Acc),gen_test'(L,[X|
                Acc]).
```

通过符号计算两个目标：

```
?-test([X1,…,Xn]).
?-test'([X1,…,Xn]).
```

并比较结果，会发现：引入累加器后，两两比较的记录与双层求和的变换类似，$\sum_{i=1}^{N}\sum_{j=i+1}^{N}x_{ij}=\sum_{j=1}^{N}\sum_{i=1}^{j-1}x_{ij}$，这个性质叫做结构可交换性。

至此看出，展开/折叠的关键问题是引入新谓词，如上例中的 gen-test'。

5. 基于逻辑的 AI 研究

长期以来，演绎在逻辑和逻辑程序设计研究中充当重要角色。近来，诱导(abduction)和归纳推理引起关注，成为新的方向。人们普遍认识到，基于逻辑的诱导和归纳是要求可扩充性(open_endedness)一类问题形式化的一个合适框架。证据如下：

·在自然语言理解中，一致文法(unification

grammer)在综合语法、语义和谈话理解方面起重要作用。

·在非单调推理方面，诸如限制和缺省推理的逻辑形式化和逻辑程序的编译被广泛研究。

·在机器学习方面，有许多基于逻辑框架的结果，如模型推理系统和逆归结(inverse resolution)。

·在类比推理方面，类比用类似于逻辑推理的形式推理规则加以自然描述，与诱导推理关系甚密。

下面叙述与上述问题相关的课题：

5.1 假言推理

Poole 等研究了假言推理的逻辑框架，认为假言推理是必需的，因为其实现简单高效，而且有足够能力实现其它形式的推理。

Inoue 开发了一个用于计算通用逻辑程序的所有可能模型的有效算法。该算法以自底向上模型生成法为基础，采用"否定即失败(Negation-by-Failure)"文字引入假言推理：一部分文字假定为 true，另一部分文字假定为 false。为了表达假定的文字，引入了模态算子，并把形如 $A_l \leftarrow A_{l+1} \wedge \cdots \wedge A_m$, not $A_{m+1} \wedge \cdots \wedge$ not A_n 的规则翻译成不包括"否定"文字的析取子句：

$A_{l+1} \wedge \cdots \wedge A_m \rightarrow$

$(NKA_{m+1} \wedge \cdots \wedge NKA_n \wedge A_l) \vee KA_{m+1} \vee \cdots \vee KA_n$.

其中 NKA 表示：A 假定为 false，KA 假定 A 为 true。通过增加下列约束把 KA 和 NKA 作为与 A 无关的新谓词：

NKA，A→for every atom A. (1)

NKA，KA→for every atom A. (2)

翻译之后，就获得了一个一阶逻辑子句集，可以使用一阶自底向上定理证明器 MGTP 来计算该集合的所有可能模型。然后只选择满足以下条件的模型 M：

对于每个基原子 A，若 KA∈M，则 A∈M. (3)

注意：①此翻译模式为原始的通用逻辑程序定义了一个编码方法。②该技术可用于计算诱导。即找到可能的假言集合，并且与给定的约束不矛盾。

5.2 类比的形式方法

类比是人类求解问题的一个重要推理方法，对于自身难以解决的问题可利用求解类似问题的知识来指导问题求解，另一方面，当没有足够的信息可以解释答案时，类比可提供较好的猜测。

实现类比推理需解决三个主要问题：1)对给定目标寻找一个合适的类比基础。2)为类比基础和目标选择可共享的重要性质。3)选择从基础通过类比映射到目标的性质。

Arima 的思想体现了上述三点。类比推理表示为推理规则：

$$\frac{S(B) \wedge P(B)}{\frac{S(T)}{P(T)}}$$

其中 T 表示目标对象，B 表示基础对象，S 是 T 和 B 之间的相似性质，P 是被映射的性质。推理规则表示：假定 T 和 B 有共同性质 S，若 B 有另一性质 P，则类比地推出 T 也有性质 P。

Arima 试图通过把表达式 $S(B) \wedge P(B)$ 修改成 $\forall x.(J(x) \wedge S(x) \supset P(x))$，将类比推理与演绎推理连结起来。其中 J(x)是为了逻辑断定 P(x)增加给 S(x)的一个假设，如果存在这样的 J(x)，类比推理就是纯粹的演绎推理。例如，学生 B 参加了乐队而忽视了学习，当得知学生 T 也参加了乐队时，我们就断定他也忽视了学习，得出此结论的理由是乐队活动繁忙。这个理由便是上述的假设的一个例子。

Arima 仔细观察类比条件，分析了 J(x)的语法结构。首先，为谓词 J 找合适的参数。假定 J 有形式 J(x,s,p)，s 和 p 分别表示类似性质和映射性质。

根据类比的特点，并不期望对象 x 和映射性质 p 之间有任何直接关系，因此 J(x,s,p)可分成两部分：$J(x,s,p) = J_{att}(s,p) \wedge J_{obj}(x,s)$，第一个成分 $J_{att}(s,p)$对应于从类比基础中抽取的信息，由于信息独立于 x 的选择，因此 $J_{att}(s,p)$与 x 无关。第二个成分 $J_{obj}(x,s)$对应于从类似中抽取的信息，因而不包括 p 参数。

对 J(x,s,p)语法结构的限制很重要。当给定目标时，它可以修剪搜索空间而找到正确的类比基础。这一功能对于把类比推理框架应用于基于情况的推理系统尤为重要。

5.3 知识表示

知识表示是 AI 研究的一个中心问题。困难之处在于没有单一的知识表示模式可以表示各种知识并同时保持应用的简单性和高效性。逻辑尽管是最有前途的候选之一，但它在表示结构化知识和变化的世界时却显得无能为力。然而，基于逻辑和逻辑程序设计的知识表示框架的开发可以解决这两个问题。

按结构化观点开发的扩展关系型数据库可以处理非标准形式，其相应语言是 CRL，它允许用户以结构化方式描述数据库。

最近出现一种新的基于逻辑的知识表示语言 Quixote，沿用了 CRL 和 CIL 的开发思想，继承了 CRL 扩充版的面向对象特征和 CIL 的部分指定项。在Quixote中，简单的数据原子、复杂的结构，甚至

FGCS的后续计划

刘海燕　张　静　编译　王献昌　审校

1.引言

为期十年的FGCS计划结束了,ICOT取得了令人瞩目的研究成果,并在FGCS'92会上展出。成果核心是FGCS原型系统,它包括并行推理机PIMS(有1000个PE,速度达100MLIPS)。在PIM上演示了许多并行软件系统:PIMOS、Kappa-P、Quixote、GDCC、MGTP和20个并行应用系统,其中法律推理系统和遗传信息分析系统倍受关注。成功的演示证明知识的并行处理行之有效,某些应用系统的N个PE可达到几乎N倍的加速!如定理证明器。总之,FGCS的科研成果得到了逻辑程序设计和并行符号处理技术领域专家的高度评价。

然而,新应用和传统计算机技术领域中的人们提出了一些非议和新的要求:有人认为FGCS的软件产品应移植到目前市场流行的计算机系统上,不仅限于PIM上的应用;也有人要求努力开发知识处理应用。这些批评和要求在十年计划期间很难处理妥当。

日本基础计算机技术开发委员会收集国内外数以百计的新计算机技术人员的观点,经近半年的讨论,决定建议MITI开展一个为期两年的后续计划,1993年四月MITI和ICOT开始了实施。

后续计划的目的是把FGCS计划中所开发的主要软件系统和基础研究成果做为开发高级计算机技术(核心是并行知识处理)的工具和支撑环境,广泛应用于研究团体。具体目标是:

1)把KL1、PIMOS和主要软件系统移植到基于UNIX的顺序系统和MIMD并行系统上。

2)采用PIMS和KL1实现的软件系统,进一步研究开发知识处理软件和并行应用软件。

目前,许多并行MIMD机进入市场,为KL1和PIMOS的移植提供了合适的体系结构和性能并且适合于ICOT开发的多种并行应用。移植为KL1软件和基于Unix的系统之间提供一个有效的软件界面,同时KL1也提供了一个非常高效的符号和知识处理的并行程序设计环境。

ICOT预备精简机构,7个实验室和规划部(近100人)减至2个研究部门和1个规划部(近40人)。总务部和国际关系部将不变(约10人),ICOT人员更替,但大多数研究组长将继续工作两年以确保研究活动继续,为传播ICOT自由软件(IFS)和进一步开发并行知识处理技术,继续寻求国际合作。

2.把并行推理系统移值到库存硬件和OS上

后续计划将开发一个可移植的、高效实现的KL1语言及其软件开发环境并把它们作为IFS公开发表,而已作为IFS发表的软件系统将移植到基于Unix的系统上。

2.1 Unix和C作为平台

由于流行Unix操作系统和C语言,所以被选择为目标系统。遗憾的是,C语言用于KL1的实现并不理想,而且Unix对于并行语言的实现也不尽人意,因为它的标准内部进程通讯机制相当弱。但考虑到可移植性,仍基本坚持Unix特征的一个非常标准的子集。尽管将可移植性放在第一位,但缺乏高效的系统是没有价值的。采用合适的目标代码设计可以合理地掩盖C语言作为一种语言实现工具的不足,通过把KL1代码编译成C来完成一个小子集的顺序实现。在Sparc Station 2上,这种实现每秒能减少

循环结构都可表示为对象标识符。

6.结论

本文总结了FGCS计划的基本研究活动,强调了逻辑程序设计研究的两个不同方向:①逻辑程序设计语言。焦点是约束逻辑程序设计和并发逻辑程序设计。②AI的基础研究和基于逻辑、逻辑程序设计的软件工程。

ICOT将在以下领域继续努力。①约束逻辑程序设计和并发逻辑程序设计的结合;②用KL1实现一个数据库;③追求诱导和归纳推理的并行实现。

主要参考文献:Koichi Furukawa, Summary of Basic Research Activities of the FGCS Project. Proceedings of the International Conference on Fifth Generation Computer Systems 1992, ICOT