# 並行論理プログラミングに基づく
# 広域分散計算パラダイムの構築

(課題番号 11680370)

平成 11 年度～平成 13 年度科学研究費補助金
(基盤研究 (C)(2))
研究成果報告書

平成 15 年 3 月

研究代表者

上田　和紀

(早稲田大学理工学部教授)

# はじめに

並行論理プログラミングは，論理プログラミング・パラダイムに情報の流れを制御するための同期機構を導入し，並行処理 (concurrency) 記述のためのプログラミング・パラダイムとしたものである．なかでも研究代表者の提案した並行論理型言語 GHC (Guarded Horn Clauses) は，意味論的に非常に簡潔な通信・同期機構をもちながら，構成が動的に変化する並行プロセス系の通信プロトコルを自然に記述できる柔軟性も併せもつ．このことから GHC は，第五世代コンピュータプロジェクトの並列核言語 KL1 のベースとしても採用された．

これまでの並行論理型言語とその処理系の研究開発は，主として均質な構造を持つ並列計算機上での高速実行を目標に行なわれてきた．その一方，コンピュータネットワークの急速な普及と共に，広域分散計算環境で稼働する効率的で安全な分散型アプリケーションを容易に構築するための方法論の確立が急務となっている．並列計算と広域分散計算はいずれも並行処理を含む概念であり，並行処理記述のための枠組を発展させることにより，広域分散環境のためのすぐれたパラダイムが構築できると期待される．そこで本研究では

(1) 非均質な計算環境下での宣言型プログラミング，
(2) 静的解析技術に基づく安全な広域分散ソフトウェア構築，

の両面から，並行論理プログラミングに基づく広域分散計算のための技術開発を行なった．

本研究によって得られた主要成果は以下の通りである．

1. **シームレスな分散実行環境の構築** — 不均質な計算ノード上で動作する複数の KL1 言語処理系をソケットを通じて透過的に接続するために，ネットワーク透過的な分散論理変数（分散単一代入チャネル）の技術要件および実装方式を検討し実現した．また，分散論理変数に対する名前付けとその解決を行なうネーミングサービスを実現した．分散単一代入チャネルは，分散プログラミングにおいて最も抽象度の高い通信手段を提供するものである．

   さらに，分散環境で不可欠となる例外処理機構の並行論理型言語処理系への導入法を検討し，実装を行った．

2. **分散実行環境における記憶管理および最適化の理論基盤** — 複数の変数によるデータ共有を静的に解析するための広義型体系である線形性体系 (linearity system) を設計，実装した．さらに，並行論理プロセスの実行を資源のやり取りの観点からとらえ直し，動的記憶管理が不要なプログラムのクラスとそれを特徴づける静的型体系—ケイパビリティ体系 (capability system)—を設計した．ケイパビリティ体系は，既存のモード体系と線形性体系とを一般化した形で統合するものである．

また，並行論理型言語の最適化コンパイルの理論基盤を与えるために，並行プロセスの挙動を定式化するインターフェース (interface) という概念を与え，さらにインターフェース解析に基づいて並行プログラムから逐次実行可能な中間コードを抽出する方法を開発した．

3. **コード移送を実現するためのインタプリタ構築技術** — 分散処理系における述語コード移送を実現するために，treecode と呼ばれる中間コード形式を設計し，またそのインタプリタを Flat GHC の純粋な機能のみを用いて記述した．さらに，Flat GHC プログラムの中間コード表現とそのインタプリタが，もとの Flat GHC プログラムと，展開・畳み込み変換によって関係付けられることを示した．

これらの研究を通じて，並行論理型言語をベースに簡明な広域分散プログラミングパラダイムを構築するための要素基盤を，理論と実装の両方を含む幅広い側面から与えることができた．

本成果報告書ではこれらの研究成果について，学会誌や国際会議の発表論文等に基づいて詳述する．

## 研究組織

研究代表者: 上田 和紀 (早稲田大学理工学部教授)
(研究協力者: 網代 育大 (早稲田大学理工学部助手，現在日本電気 (株)))
(研究協力者: 加藤 紀夫 (早稲田大学理工学部助手，大学院理工学研究科))
(研究協力者: 高木 祐介 (早稲田大学大学院理工学研究科，現在 (株) ジャステック))
(研究協力者: 松村 量　(早稲田大学大学院理工学研究科，現在富士通 (株)))
(研究協力者: 高山 啓　(早稲田大学大学院理工学研究科，現在 (株) 日立製作所)))
(研究協力者: 金木 佑介 (早稲田大学大学院理工学研究科))
(研究協力者: 粉川 友広 (早稲田大学大学院理工学研究科))

## 研究経費

(単位千円)

| | 直接経費 | 間接経費 | 合計 |
|---|---|---|---|
| 平成 11 年度 | 1,400 | 0 | 1,400 |
| 平成 12 年度 | 1,400 | 0 | 1,400 |
| 平成 13 年度 | 700 | 0 | 700 |
| 総計 | 3,500 | 0 | 3,500 |

# 研究発表

## 学会誌等論文

[1] Kazunori Ueda, Concurrent Logic/Constraint Programming: The Next 10 Years. In *The Logic Programming Paradigm: A 25-Year Perspective*, K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren (eds.), Springer-Verlag, 1999, pp. 53–71.

[2] 加藤 紀夫，上田 和紀：並行論理型言語における同期ポイントの移動の安全性について．情報処理学会論文誌：プログラミング，Vol. 41, No. SIG 2 (PRO 6) (2000年3月), pp. 13–28.

[3] Kazunori Ueda, Linearity Analysis of Concurrent Logic Programs. In *Proc. International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, 2000, pp. 253–270.

[4] Yasuhiro Ajiro and Kazunori Ueda, Kima — an Automated Error Correction System for Concurrent Logic Programs. In *Proc. Fourth International Workshop on Automated Debugging (AADEBUG 2000)*, August 2000.
`http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings.html`

[5] 網代 育大，上田 和紀：Kima: 並行論理プログラム自動修正系．コンピュータソフトウェア，Vol. 18, No. 0 (2001), pp. 122–137.

[6] 坂本 幸司，松宮 志麻，上田 和紀：並列KLIC処理系上での配列演算の最適化．情報処理学会論文誌：プログラミング，Vol. 42, No. SIG 3 (PRO 10) (2001年3月), pp. 1–13.

[7] 高木 祐介，上田 和紀：dklic: KL1による分散KL1言語処理系の実装．第4回プログラミングおよび応用のシステムに関するワークショップ (SPA2001)，日本ソフトウェア科学会，2001年3月．
`http://www.dcl.info.waseda.ac.jp/SPA2001/`

[8] 加藤 紀夫，上田 和紀：並行論理プログラムにおける逐次実行部分の抽出方法論．第3回プログラミングおよびプログラミング言語ワークショップ (PPL2001)，日本ソフトウェア科学会，2001年3月，pp. 2–13.

[9] Kazunori Ueda, Resource-Passing Concurrent Programming. In *Proc. Fourth Int. Symp. on Theoretical Aspects of Computer Software (TACS2001)*, Kobayashi, N. and Pierce, B. (eds.), Lecture Notes in Computer Science 2215, Springer-Verlag, Ocboter 2001, pp. 95–126.

[10] Yasuhiro Ajiro and Kazunori Ueda, Kima: an Automated Error Correction System for Concurrent Logic Programs. *Automated Software Engineering*, Vol. 9, No. 1 (2002), pp. 67–94.

[11] Kazunori Ueda, A Pure Meta-Interpreter for Flat GHC, A Concurrent Constraint Language. In *Computational Logic: Logic Programming and Beyond* (Essays in Honour of Robert A. Kowalski, Part I), A.C. Kakas, F. Sadri (eds.), Lecture Notes in Artificial Intelligence 2407, Springer-Verlag, 2002, pp. 138–161.

[12] Norio Kato and Kazunori Ueda, Sequentiality Analysis for Concurrent Logic Programs. In *Proc. 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002)*, Vol. 11, July 2002, pp. 329–336.

[13] Kazunori Ueda and Norio Kato, Programming with Logical Links: Design of the LMNtal Language. In *Proc. Third Asian Workshop on Programming Languages and Systems (APLAS 2002)*, 2002, pp. 115–126.

[14] 金木 佑介，加藤 紀夫，上田 和紀：KLIC 処理系における UNIX プロセス間通信を利用した例外処理の実装．第 6 回プログラミングおよび応用のシステムに関するワークショップ (SPA2003)，日本ソフトウェア科学会，2003 年 3 月．
`http://spa.jssst.or.jp/2003/program/papers/03021.pdf`

## 口頭発表

[15] Kazunori Ueda, A Close Look at Constraint-Based Concurrency (an invited tutorial). In *Proc. 17th Int. Conf. on Logic Programming (ICLP'01)*, Codognet, P. (ed.), Lecture Notes in Computer Science 2237, Springer-Verlag, November 2001, p. 9.

[16] 網代育大，上田和紀：反復深化 A* 探索によるもっともらしいプログラムの効率的な生成．人工知能学会全国大会（第 17 回）論文集，1E3-02, 2002 年 6 月．

[17] 上田和紀, 加藤紀夫：Programming with Logical Links. 日本ソフトウェア科学会第 19 回大会論文集，2002 年 9 月．

[18] 加藤紀夫，上田和紀：モード制約の漸近的一様補強による並行論理プログラムの occurs-chcek 解析．日本ソフトウェア科学会第 19 回大会論文集，2002 年 9 月．

[19] 松村量, 高山啓, 高木祐介, 加藤紀夫, 上田和紀：分散言語処理系 DKLIC の設計と実装．日本ソフトウェア科学会第 19 回大会論文集，2002 年 9 月．

[20] 上田和紀，加藤紀夫：GHC から LMNtal へ．情報処理学会 2002 年度 夏のプログラミングシンポジウム，2002 年 9 月．

## 出版物（編書）

[21] Herbert Kuchen and Kazunori Ueda (eds.), *Functional and Logic Programming—5th International Symposium on Functional and Logic Programming, FLOPS 2001.* Lecture Notes in Computer Science 2024, Springer-Verlag, March 2001.

## 受賞

[22] 上田 和紀：日本ソフトウェア科学会 第 19 回大会 高橋奨励賞，2002 (発表題目：Programming with Logical Links).

*　　*　　*

本報告書の各章の構成を簡単に記す．各章ができるだけ独立に読めるようにするため，基本的な定義等は繰返し掲げている．

第 1 章は，本研究の基礎となった並行論理プログラミング言語の歴史と展望について詳述している．本章の内容は前掲の論文 [1] に基づいている．

第 2 章からの 3 つの章は，本研究によって得られた理論的基盤である．並行論理型言語を広域分散計算に適用するときに重要な役割を演ずることになると期待される静的解析技術について述べている．

まず第 2 章は，分散実装において特に重要となる通信の線形性（受信者が単数か複数かの区別）を扱う広義の型体系と，制約概念に基づく線形性解析手法について論じている．本章の内容は前掲の論文 [3] に基づいている．

第 3 章は，並行論理プログラムにおけるプロセス間通信を資源のやりとりの観点から見直す試みと，それを支える capability system という型体系について述べている．本章の内容は前掲の論文 [9]（TACS'01 国際会議の招待論文）の改訂版に基づいている．

第 4 章は，並行論理型言語で定義した並行プロセスの中から逐次実行可能部分を見つけ出すためのインターフェース解析と呼ばれる手法を定式化し，さらにインターフェース解析に主導されて中間コードを生成する手法を提案している．本章の内容は前掲の論文 [12] に基づいている．

第 5 章からの 3 つの章は，言語設計と実装技術に関する研究成果の紹介である．

まず第 5 章では，並行論理型言語の分散実装において最も重要な技術課題である，論理変数（単一代入チャネル）の分散実装について論じている．分散単一代入変数は，異なるアーキテクチャをもつ計算ノード間の通信を，透過的かつ高い抽象度で実現するが，研究過程において多くの問題に直面した．本章は，論文 [7] や [19] などの提案をふまえてさらにさまざまな検討を加えて書かれた松村量君の修士論文 (2003) に基づいている．

第 6 章は，分散処理系において不可欠となる例外処理機構の設計と実装について論じている．並行論理型言語は通常のプログラミング言語と非常に異なるため，例外処理機構の設計と実装は困難かつ興味深い研究課題である．本章の内容は前掲の論文 [14] に基づいている．

第 7 章は，異機種間のコード移送を容易に実現するための中間コード形式 (treecode 形式) とそのインタプリタを示している．また，解釈実行される treecode の挙動を，インタプリタの展開・畳み込み (unfold/fold) 変換によって元の Flat GHC のソースプログラムと関連づける方法を例示している．本章の内容は前掲の論文 [11] に基づいている．

# 目 次

# 第1章　並行論理プログラミング言語の歴史と展望

Concurrent logic/constraint programming is a simple and elegant formalism of concurrency that can potentially address a lot of important future applications including parallel, distributed, and intelligent systems. Its basic concept has been extremely stable and has allowed efficient implementations. However, its uniqueness makes this paradigm rather difficult to appreciate. Many people consider concurrent logic/constraint programming to have rather little to do with the rest of logic programming. There is certainly a fundamental difference in the view of computation, but careful study of the differences will lead to the understanding and the enhancing of the whole logic programming paradigm by an *analytic approach*. As a model of concurrency, concurrent logic/constraint programming has its own challenges to share with other formalisms of concurrency as well. They are: (1) a counterpart of $\lambda$-calculus in the field of concurrency, (2) a common platform for various non-sequential forms of computing, and (3) type systems that cover both logical and physical aspects of computation.

## .1.1 Grand Challenges

It seems that concurrent logic programming and its generalization, concurrent constraint programming, are subfields of logic programming that are quite different from the other subfields and hence can confuse people both inside and outside the logic programming community.

While most subfields of logic programming are related to artificial intelligence in some way or other—agents, learning, constraints, knowledge bases, automated deduction, and so on—, concurrent logic/constraint programming is somewhat special in the sense that its principal connection is to concurrency.

Concurrency is a ubiquitous phenomenon both inside and outside computer systems, a phenomenon observed wherever there is more than one entity that may interact with each other. It is important for many reasons. Firstly, the phenomenon is so ubiquitous that we need a good theoretical and practical framework to deal with it. Secondly, it is concerned with the infrastructure of computing (the environment in which computer systems and programs interact with the rest of the world) as well as activities within computer systems, and as such the framework scales up. In other words, it is concerned with *computing in the large*, and accordingly, programming in the large. Thirdly, it

encompasses various important forms of non-sequential computing including parallel, distributed, and mobile computing.

Bearing this in mind, I'd like to propose the Grand Challenges of concurrent logic/constraint programming.[1] The theoretical computer science community is struggling to find killer applications, but I would claim that, as far as concurrency is concerned, there are at least three important scientific challenges besides finding killer applications.

1. *A "λ-calculus" in the field of concurrency.* It is a real grand challenge to try to have a model of concurrency and communication which is as stable as λ-calculus for sequential computation.

   We have had many proposals of models of concurrency: Petri Nets [36], Actors [2], Communicating Sequential Processes [24], and many formalisms named $X$-calculus, $X$ being Communicating Systems [29], $\pi$ [30], Action [31], Join [17], Gamma [4], Ambient [6], and so on.

   Concurrent constraint programming [38] is another important model of concurrency, though, unfortunately, it is often overlooked in the concurrency community. I proposed Guarded Horn Clauses (GHC) as a simple concurrent logic *language*, but in its first paper I also claimed:

   > "We hope the simplicity of GHC will make it suitable for a parallel computation model as well as a programming language. The flexibility of GHC makes its efficient implementation difficult compared with CSP-like languages. However, a flexible language could be appropriately restricted in order to make simple programs run efficiently. On the other hand, it would be very difficult to extend a fast but inflexible language naturally."
   >
   > — [51] (1985)

   One of the reasons why there are so many models is that there are various useful patterns of interaction, some of which are useful for high-level concurrent programming and others rather primitive. Here, a natural question arises as to whether we can find a lowest possible layer for modeling concurrency. I am not sure if people can agree upon a single common substrate, but still believe that the effort to have a simple and primitive framework (or a few of them) is very useful and will lead to higher respect of the field. Note that everybody respects λ-calculus but it still has a number of variants and some insist that it is not fully primitive (see, for example, [1]).

[1] Concurrent constraint programming can be viewed both as a generalization of concurrent logic programming and as a generalization of constraint logic programming. This article will focus on the former view since the challenges of concurrent constraint programming from the latter view should be more or less similar to those of constraint logic programming.

Needless to say, a stable calculus is a challenge but is not an ultimate goal. What we need next is a high-level programming language fully supported by a stable theory.

2. *Common platform for non-conventional computing.* The next challenge is to see if a common platform—the pair of a high-level concurrent language and an underlying theory—can be the base of various forms of non-conventional computing such as

   - parallel computing,
   - distributed/network computing,
   - real-time computing, and
   - mobile computing.

   Historically, they have been addressed by more or less different communities and cultures, but all these areas share the following property: unlike conventional sequential computing, programmers must be able to access and control the physical aspects of computation. At the same time, programmers don't want to be bothered by physical considerations in writing correct programs and porting them to different computing environments. These two requirements are referred to as *awareness* and *transparency* (of/from physical aspects).

   The fact that all these areas have to do with physical aspects means that they all have to do with concurrency. They all make sense in computing environments participated in by more than one physical entity such as 'sites' and 'devices'. This is why it is interesting to try to establish a novel unified platform for these diverse forms of non-conventional symbolic computing.

3. *Type systems and frameworks of analysis for both logical and physical properties.* The third grand challenge is a framework of static analysis to be built into concurrency frameworks. The first thing to be designed is a type system. Here I use the term "type system" in its broadest sense; that is, to have a type system means to:

   (a) design the notion of types, where the notion can be anything that is well-defined and useful either for programmers or for implementations,

   (b) define typing rules that connect the world of program text and the world of types, and

   (c) establish basic (and desirable) properties of well-typed programs such as subject reduction and strong normalization.

So a type does not necessarily represent a set of possible values a syntactic construct can denote in the standard semantics; for instance, a mode (directionality of information flow) is thought of as a type in a broad sense.

As we know, types play extremely important roles in programming languages and calculi. The fundamental difference between types and other formalisms of program analysis (such as abstract interpretation) is that, although well-typedness imposes certain constraints on allowable programs, the notion of types is exposed to programmers. Accordingly, types should be accessible to programmers and should help them understand and debug their programs better.

These features of type systems are expected to play key roles in concurrent programming. A challenge here is to deal with physical as well as logical properties of programs in a way accessible to programmers.

I believe addressing these scientific challenges is as essential as building killer applications because, only with such endeavor, declarative languages and theory-driven approach can find their raisons d'êtle.

## 1.2 Two Approaches to Addressing Novel Applications

It is natural to think that addressing novel applications requires a powerful programming language with various features. A popular approach to making a logic programming language more powerful is to generalize it or to integrate useful features into it. Constraint logic programming, inductive logic programming, higher-order logic programming, disjunctive logic programming, etc. are all such generalizations. Some extensions are better thought of as integration rather than generalization; examples are functional logic programming and multi-paradigm extensions such as Oz [46].

However, there is a totally different approach to a more powerful language, which I call an *analytic approach*. In an analytic approach, one tries to identify smaller fragments of logic programs (or of extensions of logic programs) with nice and useful properties that may lead to efficient implementation.

Note that what I mean by "powerful" here is not in terms of expressive power. By identifying possibly important fragments of a general framework and studying them carefully, one may able to establish new concepts with which one can understand the whole framework in more depth and detail. Also, one may find that some fragment allows far more efficient implementation. (A popular example where simplicity is the source of efficiency is the RISC architecture.) One may build programming tools that take advantage of the properties of fragments. They are a source of power because it may open up new application areas that could not be addressed by the general framework.

The above claim could be understood also from the following analogy: having a notion of Turing machines does not necessarily mean that we don't have to study pushdown

or finite-state automata. They have their values in their own rights. Another example is the relationship between untyped and typed $\lambda$-calculi. Yet another obvious example is the identification of Horn sentences from full first-order formulae, without which the logic programming paradigm would not exist today. Examples of smaller fragments of logic programming languages that have been studied in depth are Datalog (no function symbols) and concurrent logic languages (no search in exchange of reactiveness).

The analytic approach is useful also when one attempts to generalize or integrate features. Integration will succeed only after the features to be integrated have been well understood and the interface between them has been carefully designed. A criterion of success is whether one can give clean semantics to the whole integrated framework as well as to each component. If the components interact only at the meta (or extralogical) level, the whole framework is considerably more complicated (in terms of semantics) than their components, which means the verification and manipulation of programs become considerably harder. This issue will be discussed in the next section.

## 1.3 Logic Programming vs. Concurrent Logic Programming

Concurrent logic programming was born from the study of concurrent execution of logic programs. It turned out to enjoy a number of nice properties both as a formalism and as a language for describing concurrency. In the logic programming community, however, concurrent logic programming has always been a source of controversy. Unfortunately, the controversy was by and large not very technical and did not lead to deeper understanding of the paradigms.

A typical view of concurrent logic programming has been:

$$\begin{aligned} \text{Concurrent LP} &= \text{LP} + \text{committed choice} \\ &= \text{LP} - \text{completeness} \end{aligned}$$

Although both the first and the second equations are not totally wrong, viewing committed choice simply as losing completeness is too superficial.

Committed choice or don't-care nondeterminism is an essential construct in modeling reactive computing and has been studied in depth in the concurrency community. It is essential because one must be able to model a process or an agent that performs *arbitration.* (For instance, a receptionist will serve whoever *(s)he thinks* comes first, rather than whoever comes first.) Semantically, it is much more than just discarding all but one of possible execution branches. All the subtleties lie in what events or information should be the basis of choice operations; in other words, the subtleties lie in the semantics of guard rather than the choice itself. The presence or absence of nondeterminism makes fundamental difference to the denotational semantics of concurrency (see [24] for example). When I was designing Guarded Horn Clauses, I believed don't-

care nondeterminism was so essential that it was a bad idea to retain both don't-care nondeterminism and don't-know nondeterminism in a single model of computation.

Nevertheless, we also found in our experiences with concurrent logic languages that most of the predicates are deterministic and very few predicates perform arbitration—even though they change the whole semantical framework. Thus the aspect of arbitration is not to be overstated. A much more productive view of concurrent logic programming will accordingly be:

$$\begin{aligned} \text{Concurrent LP} &= \text{LP} + \text{directionality of dataflow} \\ &= \text{LP} + \text{embedded concurrency control} \end{aligned}$$

This is more productive because it emphasizes the aspect of dataflow synchronization, an important construct also in logic programming without committed choice. Examples where dataflow synchronization plays important roles include delaying, coroutining, sound negation-as-failure, and the Andorra principle [37]. Dataflow-centered view of the execution logic programs best captures the essence of concurrent logic/constraint programs, as became clear from the *ask* + *tell* formulation advocated by Saraswat [38].

Another reason why the above view is more productive is that it addresses mode systems that prescribe the directionality of dataflow. Mode systems are attracting more interest in various subfields of logic programming because

- it shares with type systems many good properties from which both programmers and implementations can benefit, and
- many (if not all) predicates we write have a single intended mode of use, and there are a lot of situations where this fact can be exploited in interesting ways.

For example, Mercury [47] takes full advantage of strong moding to yield very efficient code.[2] Inductive logic programming benefits from moding in reducing search space [32]. Concurrent logic/constraint programming benefits enormously from strong moding both in implementation and programming [61, 63, 13, 3], and I strongly believe that

$$\text{Moded Concurrent LP} = \text{ask} + \text{tell} + \text{strong moding}$$

is one of the most flexible realistic models of concurrency.

It is vital to see that ordinary logic programming and concurrent logic programming are targeted at different scopes. What logic programming is concerned with include knowledge representation, reasoning, search, etc., while concurrent logic programming aims at a simple programming and theoretical model of concurrency and communication. Accordingly, concurrent logic languages should aim at general-purpose algorithmic

[2]Note, however, that the mode system of Mercury is very different from the mode system of Moded Flat GHC discussed in this paper; the former deals with the change of instantiatedness, which is a temporal property, while the latter deals with polarity, which is a non-temporal property [63].

languages which can potentially act as coordinators of more application-specific logic languages.

Since the conception of concurrent logic programming, how to reconcile two essential features in parallel knowledge information systems, search and reactiveness, has been one of the most difficult problems. The two paradigms could be integrated but should be done with utmost care. The solution adopted in PARLOG [14] was to use all-solutions predicates (à la `findall` in Prolog) to interface between the world of don't-know nondeterminism and the world of don't-care nondeterminism. The key issue here is how to gather multiple solutions obtained from different binding environments into a single data structure. Whether the all-solutions construct can be given clean, declarative meaning and whether it allows efficient implementation depend on the program and the goal for which solutions are to be collected [33, 55]. Roughly speaking, the requirement has to do with the proper treatment of logical variables. Existing all-solutions predicates in Prolog involve the copying of solutions, exhibiting *impedance mismatch.*[3]

Moding seems to play an important role here; we conjecture that all-solutions predicates can be given simple, object-level semantics if the program and the goal are well-moded under an appropriate mode system similar to the mode system for Moded Flat GHC [61].

Another example where moding played an important role in essentially the same way is the First Order Compiler [41], a compiler from a class of full first-order formulae into definite clauses.

The issue of clean interfacing arises in constraint logic programming systems also. In realistic applications of constraint satisfaction, it is often crucial that a constraint solver can run concurrently with its caller so that the latter be able to observe and control the behavior of the former incrementally. However, language constructs for doing so are yet to be refined.

## 1.4 An Application Domain: Parallel/Network Programming

Where should concurrent logic/constraint programming languages find promising applications? I believe that the most important areas to address are parallel and network applications for a number of reasons:

1. Even "modern" languages like Obliq and Java feature rather classical concurrency

[3] Oz features *computation spaces* (the pair of a local constraint store and a set of processes working on the store) as first-class citizens, with which encapsulated search can be programmed as higher-order combinators [45]. This approach is certainly cleaner in that a set of solutions is represented using (procedures returning) computation spaces instead of copied terms. In an analytic approach, however, we are interested in identifying a class of programs and goals for which a set of solutions can be represented without using higher-order constructs.

constructs such as monitors and explicit locking. In more traditional languages like C or Fortran, parallel/network programming is achieved with APIs such as MPI, Unix sockets, and POSIX threads. These constructs are all low-level compared with synchronization based on dataflow and arbitration based on choice, and programming with APIs seems to be a step backwards from writing provably correct programs even though verification is not impossible.

2. Parallel computing and distributed computing are considerably more difficult than sequential computing. Good models and methodologies to build large applications quickly are desperately called for.

3. These areas are becoming increasingly popular in a strong trend towards large-scale global computing environments both for high-performance distributed computing [16] and for virtual network communities such as Virtual Places, Community Places and Matrix.[4]

4. These areas give us a good opportunity to demonstrate the power of small and yet "usable" languages with an appropriate level of abstraction. It seems essential to keep the languages simple enough—and much simpler than Java—to be amenable to theoretical treatment and to make them as easy to learn as possible. I anticipate that amenability to theoretical treatment, if well exploited, will be of enormous practical importance in these areas.

From a language point of view, the last point is the most challenging. Consider writing secure network applications with mobile code. This involves various requirements:

- Specification of *physical locations* (sites) where computation should take place.
- Reasoning about *resources*, such as time, stack and heap, that the computation may consume. Without it, downloaded code might make a so-called DoS (denial of service) attack by monopolizing computation resources.
- Security at various levels. Some of the high-level properties such as consistency of communication protocols can be guaranteed by typing and moding. Other high-level security issues may require more sophisticated analysis and verification. Low-level security could partly be left to Java's bytecode verifier if we use Java or Java bytecode as a target language.
- Transmission of various entities across possibly heterogeneous platforms. In addition to program code, linked data structures and symbols (usually given unique IDs locally on each site) are the main points of consideration in symbolic languages.

[4]Interestingly, the designers of Grid [16], Virtual Places, Community Places and Matrix have all worked actively on concurrent logic/constraint programming.

The requirements are so complicated and diverse that addressing them in an ad hoc way would result in a theoretically intractable language. It is an interesting and big challenge to obtain a language that allows and encourages formal reasoning about physical and logical properties of programs.

It may be a good idea for declarative language communities to share a set of (more concrete) challenges of the form "how can we program $X$ in our formalisms?" to facilitate comparison between different paradigms. Instances of $X$ may be:

- dynamic data structures (e.g., cyclic graphs; most declarative languages just ignore them, regrettably),
- live access counters of WWW pages,
- teleconferencing, and
- MUD (multi-user dungeon; text-based virtual reality).

## 1.5 Experiences with Guarded Horn Clauses and KL1

Although the progress has been admittedly slow, I am quite optimistic about the future of concurrent logic/constraint programming. My optimism is based on 15 years of our experiences with the paradigm since the initial stage of the Japanese Fifth Generation Computer Systems (FGCS) project. Figures 1–2 show the history of Guarded Horn Clauses (GHC) and KL1 as well as related events. The role and the history of the kernel language in the FGCS project are discussed in detail in my article in [44].

### 1.5.1 GHC as the Weakest Fragment of Concurrent Constraint Programming

After 13 years of research, heated discussions and programming experiences since the proposal of GHC, it turned out that this simplest fragment of concurrent constraint programming was surprisingly stable and versatile.

Let us see why it was so stable. GHC is thought of as the weakest Concurrent Constraint Language in the following senses: First, it features *ask* and *eventual tell* (i.e., publication of constraints *after* committed choice) but not *atomic tell* (publication *upon* committed choice). Second, its computation domain is a set of finite trees. Nevertheless, GHC as well as its ancestors featured fundamental constructs for a concurrent programming language from the very beginning:

- parallel composition,
- creation of local variables,
- nondeterministic (committed) choice,

- 1983 Concurrent Prolog [42] and initial version of PARLOG [14]
- 1983-84 Big controversy (inside ICOT) on LP vs. concurrent LP for parallel knowledge information processing systems [44]
- 1985 First paper on GHC [51]
- 1985 GHC-to-Prolog compiler [52] used in our initial experiments
- 1985–86 GHC considered too general; subsetted to Flat GHC
- 1986 Prolog-to-GHC compiler performing exhaustive search [53]
- 1987 MRB (1-bit reference counting) scheme for Flat GHC [9]
- 1987 First parallel implementation of Flat GHC on Multi-PSI v1 (6 PEs) [25]
- 1987 ALPS [28] gave a logical interpretation of communication primitives
- 1987–1988 KL1 designed based on Flat GHC, the main extension being the *shoen* construct [57]
- 1988 Parallel implementation of KL1 on Multi-PSI v2 (64 PEs) [34]
- 1988 Strand [15] (evolved later into PCN [7] and CC++ [8])
- 1988 PIMOS operating system [10]
- 1988 Unfold/fold transformation and transaction-based semantics [54]
- 1989 Concurrent Constraint Programming [38]
- 1989 Controversy on atomic vs. eventual tell (Kahn's article in [44])
- 1989 MGTP (Model Generation Theorem Prover in KL1) project [19]

Fig. 1.1: GHC, KL1, and related events (Part I)

- 1989 Message-oriented implementation of Flat GHC [58]
- 1990 Mode systems for Flat GHC [58]
- 1990 Structural operational semantics for Flat GHC [59]
- 1990 Janus [39]
- 1991 Denotational semantics of CCP [40]
- 1991 AKL [26] (later evolved into Oz, Oz2, and Oz3)
- 1992 Parallel implementation of KL1 on PIM/m and PIM/p [48, 23]
- 1992 Various parallel applications written in KL1, including OS, biology, CAD, legal reasoning, automated deduction, etc. [11, 22, 35]
- 1992 Message-oriented parallel implementation of Moded Flat GHC [60]
- 1992 KLIC (KL1-to-C compiler) designed [12]
- 1992 MGTP solved an open problem (IJCAI'93 award) [20]
- 1994 Proof system for CCP [5]
- 1994 Moded Flat GHC formulated in detail [61]
- 1994 ToonTalk, a visual CCP language [27]
- 1995 Constraint-based mode systems put into practice [63]
- 1996 klint, a mode analyzer for KL1 programs
- 1996 Strong moding applied to constraint-based error diagnosis [13]
- 1997 kima, a diagnoser of ill-moded programs
- 1997 KLIEG, a visual version of KL1 and its programming environment [50]
- 1997 Strong moding applied to constraint-based error correction [3]

Fig. 1.2: GHC, KL1, and related events (Part II)

- value passing, and
- data structures (trees, lists, etc.).

The last two points are in contrast with other models of concurrency such as CCS and (theoretical) CSP that primarily focused on atomic *events*. GHC was proposed primarily as a concurrent language, though it was intended to be a model as well (Section 1.1).

Furthermore, GHC as well as its ancestors had the following features from the beginning:

1. *Reconfigurable process structures.* Concurrent logic languages supported dynamic reconfiguration of interprocess communication channels and dynamic creation of processes. Most mathematical models of concurrency, on the other hand, did not feature reconfigurable process structures until $\pi$-calculus was proposed in late 1980's.

2. *Object (process) identity.* This is represented by logical variables (occurring as the arguments of processes) through which processes interact with each other. Although objects themselves are not first-class in GHC, the variables identifying processes can be passed around to change the logical configuration of processes. Hence the processes were effectively mobile exactly in the sense of mobile processes in $\pi$-calculus.

3. *Input/output completely within the basic framework.* The input/output primitives of "declarative" languages had generally been provided as marginal constructs and in a quite unsatisfactory manner. I thought the design of general-purpose languages should proceed in the opposite way *by taking the semantics of input/output constructs as a boundary condition of language design.* Concurrent logic programs are often thought of as less declarative than logic programs, but real-life concurrent logic programs projected to logic programs (by forgetting synchronization) are much more declarative than real-life Prolog programs.

Later on, several important features were added:

1. KL1 [57] featured the notion of physical locations, though in a primitive form, to allow programmers to describe load balancing of parallel computation.

2. The mode system [61] introduced the notion of (statically decidable) *read/write capabilities* or *polarities* into each variable occurrence and each position of (possibly nested) data structures. In well-moded programs, a write capability can be passed around but cannot be copied or discarded, while a read capability can be copied and discarded. Well-modedness can be established by constraint-based mode analysis which is essentially a unification problem over feature graphs.

3. Linearity analysis, which distinguishes between one-to-one and one-to-many communication [64], enabled compile-time garbage collection and turned out to play a key role in parallel/distributed symbolic computation. For instance, parallel operations on an array in shared memory can be done without any interference by splitting the array into pieces in-place, letting parallel processes operate on its own piece in-place, and then merging the resulting pieces in-place [62]. Both mode analysis and and linearity analysis support resource-conscious programming by being sensitive to the number of occurrences of variables.

### 1.5.2 Logical Variables as Communication Channels

Most of the outstanding features of concurrent logic/constraint languages come from the power and the flexibility of logical variables as communication channels. Logical variables support:

- data- and demand-driven communication,
- messages with reply boxes,
- first-class channels (encoded as lists or difference lists),
- replicable read-only data, and
- implicit redirection across sites.

It is surprising that all these features are supported by a single mechanism. This uniformity gives tremendous benefits to theoretical foundations and programming systems, since a single framework can cover all these features.

### 1.5.3 Evolution as Devolution

It is generally understood that concurrent logic programming evolved into concurrent constraint programming by ALPS's logical interpretation of communication primitives [28] and Saraswat's reformulation of concurrent logic programming as a framework of concurrency [38]. However, I have an impression that the role of concurrent constraint programming as a generalization of concurrent logic programming has been a bit different from the role of constraint logic programming as a generalization of logic programming. While constraint logic programming has found several useful domains and applications, the main contribution of concurrent constraint programming has been in the understanding of the essence of the framework and the promotion of the study of semantics. The set of useful general-purpose constraint systems (other than obvious ones such as finite trees, integers and floating-point numbers) for concurrent constraint programming is yet to be identified.

Indeed, the history of the practice of concurrent logic programming could be summarized as "evolution by devolution" [49]. As conjectured in [51] (quoted in Section 1.1), GHC as the weakest fragment of concurrent constraint programming (d)evolved first by disallowing nested guards (Flat GHC) and then by featuring a mode system. Virtually all programs now written in KL1 and run by KLIC [12] are well-moded, though KLIC currently does not support mode analysis. On the other hand, there are only a few constructs added to KL1: *shoen* (a Japanese word meaning 'manor') as a unit of observing and controlling computation, the `@node( )` construct for process migration, and priorities.

Strong moding has degenerated unification (a case of constraint solving) to assignment to a variable, but has made GHC a much securer concurrent language; that is, it guarantees that constraints to be published to a shared store (binding environment) are always consistent with the current store. Linearity analysis guarantees that some class of programs (including most sorting programs, for instance) can run without generating garbage. Both analyses are useful not only for efficient implementation but also for the precise analysis of computational cost, which is essential in real-time computing and network programming with mobile code. In this way, degeneration may find new applications which could not be addressed by more general languages.

Oz [46, 21] has taken a totally different approach from GHC. It has incorporated a number of new constructs such as ports (a primitive for many-to-one communication), cells (containers of values that allow destructive update), computation space (encapsulated store, somewhat affected by nested guards of full GHC and KL1's *shoen*), higher-order, etc., and has moved from fine-grained to coarse-grained concurrency. It still encompasses a concurrent constraint language, but is now better viewed as a multi-paradigm language.

In contrast, I'd like to keep GHC a *pure* concurrent constraint language. (Moded Flat) GHC is quite a small fragment but it is yet to be seen what additional constructs are really necessary to make pure concurrent logic/constraint languages usable.

## 1.6 Some Failures and Problems

Although I'm optimistic about its future technically, concurrent logic/constraint programming has experienced a number of non-technical problems.

1. *Misleading names of the paradigms.* Concurrent logic languages are primarily *concurrent* programming languages though they retain the nice properties of logic programming wherever possible, such as soundness of proof procedures and declarative reading.[5] Unfortunately, concurrency is so unpopular in the logic programming community that concurrent logic programming often sounds like

[5] One may argue that whether a concurrent language retains nice properties of logic programming is not very important, but this is not true. This criterion worked as a strong guideline of language design and resulted in many desirable properties as a concurrent language.

nothing more than an incomplete variant of logic programming. (An even worse name once used was *committed-choice languages.*)

Concurrent constraint programming (languages) sounds better in this respect, but it has another problem. The conception of concurrent constraint programming is often said to date from ALPS, but this often results in the ignorance of its pre-history, the era of concurrent logic programming. *Concurrent logic languages are, by definition, (instances of) concurrent constraint languages.*[6]

2. *Community problem.* Although concurrent constraint programming is an elegant formalism of concurrency, it was born from logic programming, a paradigm quite unpopular in the concurrency community and the community of concurrent programming. So, this important paradigm can very easily be forgotten by both the logic programming community and the communities of concurrency theory and concurrent programming!

3. *Shortage of communication with neighboring communities.* This is a very general tendency and unfortunately applies to various communities related to concurrency. There are many techniques independently invented in the functional programming community, the (concurrent) object-oriented programming community and the (concurrent) logic programming community. Declarative arrays, frameworks of program analysis, scheduling of fine-grained tasks, and distributed memory management are all such examples.

   A bit more technical reason why concurrent logic/constraint programming is still unpopular in the concurrency community at large may be that its formulation looks rather indirect—popular and mundane idioms of concurrent programming such as objects, messages, and channels are all encoded entities.

4. *Few research groups.* Except for research groups on semantics, there are only two active (virtual) groups working on languages and implementation; one is the group working on Oz and the other working on GHC/KL1. Many key people who founded the field "graduated" too early before the paradigm became well understood and ready to find interesting applications. However, a good news is that most of them are working on the potential applications of the paradigm discussed earlier in this article. This leaves us a challenge to bridge the gap between what the paradigm offers and what the applications require.

[6] At an early stage, I had understood GHC computation in terms of the exchange of bindings between processes rather than a restricted proof procedure:

> "... 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."
>
> — [56] (1986)

Of course, it was definitely after the proposal of concurrent constraint programming that binding-(constraint-) centered view became popular and the study of semantics made progress.

5. *Textbooks.* Good textbooks and tutorial materials are yet to be published. There are some on concurrent logic programming, such as Shapiro's survey [43], but a tutorial introduction to the semantical foundations is still awaited. It's time to recast important concepts and results scattered over technical papers and represent them from the current perspective.

## 1.7 Conclusions

Concurrent logic/constraint programming has been a simple and extremely stable formalism of concurrency, and at the same time it has been a full-fledged programming language. It is unique among many other proposals of concurrency formalisms in that information, communication and synchronization are modeled in terms of constraints, a general and mathematically well-supported framework. It is unique also in that its minimal framework (with *ask*, *eventual tell*, parallel composition, guarded choice and scoping) is almost ready for practical programming. That is, there is little gap between theory (computational model) and practice (programming language). The stability of the core concurrent constraint programming with *ask*+ *eventual tell* indicates that the logic programming community came up with something essential in early 1980's and noticed it in full in late 1980's.

There seems to be a feeling that concurrent logic/constraint programming has established an independent scientific discipline *outside* the logic programming paradigm. There is certainly a fundamental difference in how computation is viewed—one is on deduction while the other is on reactive agents. However, the fact that each paradigm features what the other does not have and the fact that they still share a lot of technicalities at less fundamental levels strongly indicate that they should benefit from each other. An interesting example of such bridging can be found in Constraint Handling Rules [18], a concurrent constraint language specifically designed for programming constraint systems.

Since concurrent constraint languages aim at general-purpose languages, they can benefit from static analysis more strongly than logic programming languages can. So I'd like to conclude this article by claiming that constraint-based static analysis can make concurrent constraint programming a simple, powerful, and safe language for

- parallel and high-performance computing,
- distributed and network computing, and
- real-time and mobile computing.

Its role in concurrent constraint programming is analogous to, but probably more than, the role of type systems in $\lambda$-calculus.

## References


[1] Abadi, M., Cardelli, L., Curien, P.-L. and Lévy, J.-J., Explicit substitutions. *J. Functional Programming*, Vol. 1, No. 4 (1991), pp. 375–416.

[2] Agha, G. A., *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, MA, 1986.

[3] Ajiro, Y., Ueda, K. and Cho, K., Error-correcting Source Code. In *Proc. Fourth Int. Conf. on Principles and Practice of Constraint Programming (CP98)*, LNCS 1520, Springer-Verlag, Berlin, 1998, pp. 40–54.

[4] Banâtre, J.-P. and Le Métayer, D., The GAMMA Model and Its Discipline of Programming. *Science of Computer Programming*, Vol. 15, No. 1 (1990), pp. 55–77.

[5] de Boer, F. S., Gabbrielli, M., Marchiori, E. and Palamidessi, C., Proving Concurrent Constraint Programs Correct. In *Conf. Record of the 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, ACM Press, 1994, pp. 98–108.

[6] Cardelli, L. and Gordon, A. D., Mobile Ambients. In *Foundations of Software Science and Computational Structures*, Maurice Nivat (ed.), LNCS 1378, Springer-Verlag, Berlin, 1998, pp. 140–155.

[7] Chandy, K. M. and Taylor, S., *An Introduction to Parallel Programming.* Jones and Bartlett, Boston, 1992.

[8] Chandy, K. M. and Kesselman, C., CC++: A Declarative Concurrent Object-Oriented Programming Notation. In *Research Directions in Concurrent Object-Oriented Programming*, Agha, G., Wegner, P. and Yonezawa, A. (eds.), The MIT Press, Cambridge, MA, 1993, pp. 281–313.

[9] Chikayama, T. and Kimura, Y., Multiple Reference Management in Flat GHC. In *Proc. 4th Int. Conf. on Logic Programming (ICLP'87)*, The MIT Press, Cambridge, MA, 1987, pp. 276–293.

[10] Chikayama, T., Sato, H. and Miyazaki, T., Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, Tokyo, 1988, pp. 230–251.

[11] Chikayama, T., Operating System PIMOS and Kernel Language KL1. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha and IOS Press, Tokyo, 1992, pp. 73–88.

[12] Chikayama, T., Fujise, T. and Sekita, D., A Portable and Efficient Implementation of KL1. In *Proc. 6th Int. Symp. on Programming Language Implementation and*

*Logic Programming (PLILP'94)*, LNCS 844, Springer-Verlag, Berlin, 1994, pp. 25–39.

[13] Cho, K. and Ueda, K., Diagnosing Non-Well-Moded Concurrent Logic Programs, In *Proc. 1996 Joint Int. Conf. and Symp. on Logic Programming (JICSLP'96)*, The MIT Press, Cambridge, MA, 1996, pp. 215–229.

[14] Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.

[15] Foster, I. and Taylor, S., Strand: a Practical Parallel Programming Tool. In *Proc. 1989 North American Conf. on Logic Programming (NACLP'89)*, The MIT Press, Cambridge, MA, 1989, pp. 497–512.

[16] Foster, I. and Kesselman, C., *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann, San Francisco, 1998.

[17] Fournet, C., Gonthier, G. Lévy, J.-J., Maranget, L. and Rémy, D., A Calculus of Mobile Agents. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, LNCS 1119, Springer-Verlag, Berlin, 1996, pp. 406–421.

[18] Frühwirth, T., Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.

[19] Fujita, H. and Hasegawa, R., A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm. In *Proc. Eighth Int. Conf. on Logic Programming (ICLP'91)*, The MIT Press, Cambridge, MA, 1991, pp. 535–548.

[20] Fujita, M., Slaney, J. and Bennett, F., Automatic Generation of Some Results in Finite Algebra. In *Proc. 13th Int. Joint Conf. on Artificial Intelligence (IJCAI'93)*, 1993, pp. 52–57.

[21] Haridi, S., Van Roy, P., Brand, P. and Schulte, C., Programming Languages for Distributed Applications. *New Generation Computing*, Vol. 16, No. 3 (1998), pp. 223–261.

[22] Hasegawa, R. and Fujita, M., Parallel Theorem Provers and Their Applications. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha and IOS Press, Tokyo, 1992, pp. 132–154.

[23] Hirata, K., Yamamoto, R., Imai, A., Kawai, H., Hirano, K., Takagi, T., Taki, K., Nakase, A. and Rokusawa, K., Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha and IOS Press, Tokyo, 1992, pp. 436–459.

[24] Hoare, C. A. R., *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[25] Ichiyoshi N., Miyazaki T. and Taki, K., A Distributed Implementation of Flat GHC on the Multi-PSI. In *Proc. 4th Int. Conf. on Logic Programming (ICLP'87)*, The MIT Press, Cambridge, MA, 1987, pp. 257–275.

[26] Janson, S. and Haridi, S., Programming Paradigms of the Andorra Kernel Language. In *Proc. 1991 Int. Logic Programming Symp. (ILPS'91)*, The MIT Press, Cambridge, MA, 1991, pp. 167–183.

[27] Kahn, K. M., ToonTalk—An Animated Programming Environment for Children. *J. Visual Languages and Computing*, Vol. 7, No. 2 (1996), pp. 197–217.

[28] Maher, M. J., Logic Semantics for a Class of Committed-Choice Programs. In *Proc. Fourth Int. Conf. on Logic Programming (ICLP'87)*, The MIT Press, Cambridge, MA, 1987, pp. 858–876.

[29] Milner, R., *Communication and Concurrency*. Prentice-Hall International, London, 1989.

[30] Milner, R., Parrow, J. and Walker, D., A Calculus of Mobile Processes, I+II. *Information and Computation*, Vol. 100, No. 1 (1992), pp. 1–77.

[31] Milner, R., Calculi for Interaction. *Acta Informatica*, Vol. 33, No. 8 (1996), pp. 707–737.

[32] Muggleton, S., Inverse Entailment and Progol. *New Generation Computing*, Vol. 13 (1995), pp. 245–286.

[33] Naish, L. All Solutions Predicates in Prolog. In *Proc. 1985 Symp. on Logic Programming (SLP'85)*, IEEE, 1985, pp. 73–77.

[34] Nakajima K., Inamura Y., Rokusawa K., Ichiyoshi N. and Chikayama, T., Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. Sixth Int. Conf. on Logic Programming (ICLP'89)*, The MIT Press, Cambridge, MA, 1989, pp. 436–451.

[35] Nitta, K., Taki, K. and Ichiyoshi, N., Experimental Parallel Inference Software. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha and IOS Press, Tokyo, 1992, pp. 166–190.

[36] Petri, C.A., Fundamentals of a Theory of Asynchronous Information Flow. In *Proc. IFIP Congress 62*, North-Holland Pub. Co., Amsterdam, 1962, pp.386–390.

[37] Santos Costa V., Warren, D. H. D. and Yang, R., Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Proc. Third ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP'91)*, SIGPLAN Notices, Vol. 26, No. 7 (1991), pp. 83–93.

[38] Saraswat, V. A. and Rinard, M., Concurrent Constraint Programming (Extended Abstract). In *Conf. Record of the Seventeenth Annual ACM Symp. on Principles of Programming Languages*, ACM Press, 1990, pp. 232–245.

[39] Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conference on Logic Programming (NACLP'90)*, The MIT Press, Cambridge, MA, 1990, pp. 431–446.

[40] Saraswat, V. A., Rinard, M. C. and Panangaden, P., Semantic Foundations of Concurrent Constraint Programming. In *Conf. Record of the Eighteenth Annual ACM Symp. on Principles of Programming Languages*, ACM Press, 1991, pp. 333–352.

[41] Sato, T. and Tamaki, H., First Order Compiler: A Deterministic Logic Program Synthesis Algorithm. *J. Symbolic Computation*, Vol. 8, No. 6 (1989), pp. 605–627.

[42] Shapiro, E. Y., Concurrent Prolog: A Progress Report. *IEEE Computer*, Vol. 19, No. 8 (1986), pp. 44–58.

[43] Shapiro, E., The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.

[44] Shapiro, E. Y., Warren, D. H. D., Fuchi, K., Kowalski, R. A., Furukawa, K., Ueda, K., Kahn, K. M., Chikayama, T. and Tick, E., The Fifth Generation Project: Personal Perspectives. *Comm. ACM*, Vol. 36, No. 3 (1993), pp. 46–103.

[45] Schulte, C. and Smolka, G., Encapsulated Search for Higher-order Concurrent Constraint Programming. In *Proc. 1994 International Logic Programming Symp. (ILPS'94)*, The MIT Press, Cambridge, MA, 1994, pp. 505–520.

[46] Smolka, G., The Oz Programming Model. In *Computer Science Today*, van Leeuwen, J. (ed.), LNCS 1000, Springer-Verlag, Berlin, 1995, pp. 324–343.

[47] Somogyi, Z., Henderson, F. and Conway, T., The Execution Algorithm of Mercury, An Efficient Purely Declarative Logic Programming Language. *J. Logic Programming*, Vol. 29, No. 1–3 (1996), pp. 17–64.

[48] Taki, K., Parallel Inference Machine PIM. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha and IOS Press, Tokyo, 1992, pp. 50–72.

[49] Tick, E. The Deevolution of Concurrent Logic Programming Languages. *J. Logic Programming*, Vol. 23, No. 2 (1995), pp. 89–123.

[50] Toyoda, M., Shizuki, B., Takahashi, S., Matsuoka, S. and Shibayama, E., Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment. In *Proc. IEEE Symp. on Visual Languages*, IEEE, 1997, pp. 76–83.

[51] Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming '85*, Wada, E. (ed.), LNCS 221, Springer-Verlag, Berlin, 1986, pp. 168–179.

[52] Ueda, K. and Chikayama, T., Concurrent Prolog Compiler on Top of Prolog. In *Proc. 1985 Symp. on Logic Programming (SLP'85)*, IEEE, 1985, pp. 119–126.

[53] Ueda, K. Making Exhaustive Search Programs Deterministic. In *Proc. Third Int. Conf. on Logic Programming (ICLP'86)*, LNCS 225, Springer-Verlag, Berlin, 1986, pp. 270–282. Revised version in *New Generation Computing*, Vol. 5, No. 1 (1987), pp. 29–44.

[54] Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, Tokyo, 1988, pp. 582–591.

[55] Ueda, K., Parallelism in Logic Programming. In *Information Processing 89, Proc. IFIP 11th World Computer Congress*, North-Holland/IFIP, 1989, pp. 957–964.

[56] Ueda, K., Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. ICOT Tech. Report TR-208, ICOT, Tokyo, 1986. Also in *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, Amsterdam, 1988, pp. 441–456.

[57] Ueda, K. and Chikayama, T. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.

[58] Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming (ICLP'90)*, The MIT Press, Cambridge, MA, 1990, pp. 3–17. Revised version in *New Generation Computing* [61].

[59] Ueda, K., Designing a Concurrent Programming Language. In *Proc. InfoJapan'90*, Information Processing Society of Japan, Tokyo, 1990, pp. 87–94.

[60] Ueda, K. and Morita, M., Message-Oriented Parallel Implementation of Moded Flat GHC. *New Generation Computing*, Vol. 11, No. 3–4 (1993), pp. 323–341.

[61] Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.

[62] Ueda, K., Moded Flat GHC for Data-Parallel Programming. In *Proc. FGCS'94 Workshop on Parallel Logic Programming*, ICOT, Tokyo, 1994, pp. 27–35.

[63] Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, LNCS 1068, Springer-Verlag, Berlin, 1996, pp. 134–153.

[64] Ueda, K., Linearity Analysis of Concurrent Logic Programs. In *Proc. International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, 2000, pp. 253–270.

# 第2章　並行論理プログラムの線形性解析

Automatic memory management and the hiding of the notion of pointers are the prominent features of symbolic processing languages. They make programming easy and guarantee the safety of memory references. For the memory management of linked data structures, copying garbage collection is most widely used because of its simplicity and desirable properties. However, if certain properties about runtime storage allocation and the behavior of pointers can be obtaind by static analysis, a compiler may be able to generate object code closer to that of procedural programs. In the fields of parallel, distributed and real-time computation, it is highly desirable to be able to identify data structures in a program that can be managed without using garbage collection. To this end, this paper proposes a framework of linearity analysis for a concurrent logic language Moded Flat GHC, and proves its basic property. The purpose of linearity analysis is to distinguish between fragments of data structures that may be referenced by two or more pointers and those that cannot be referenced by two or more pointers. Data structures with only one reader are amenable to compile-time garbage collection or local reuse. The proposed framework of linearity analysis is constraint-based and involves both equality and implicational constraints. It has been implemented as part of klint v2, a static analyzer for KL1 programs.

## 2.1 Introduction

In whatever programming language, variables can be viewed as a means of communication as well as a means of storage. When viewed as a means of communication,

- assigning a value to a variable at some point in a time space amounts to sending, and
- reading the value of a variable at another point in the time space amounts to receiving.

A value once assigned is usually read at least once before it is altered by subsequent assignments.[1] The communication is one-to-one when the value is read exactly once, while it is one-to-many when the value is read more than once.

When the value to be communicated is non-atomic, it is usually created on a heap and a variable holds a pointer to it rather than the value itself. The 'final' reader of

[1] In the case of single-assignment variables, the value once assigned will not be altered forever.

$$
\begin{aligned}
\text{(Program)}\ P &::= \text{set of } C && (1)\\
\text{(Program Clause)}\ C &::= A \text{ :- | } B && (2)\\
\text{(Body)}\ B &::= \text{multiset of } G && (3)\\
\text{(Goal)}\ G &::= T_1 = T_2 \ \mid\ A && (4)\\
\text{(Non-unification Goal)}\ A &::= p(T_1, \ldots, T_n), \\
&\qquad p \text{ is a predicate other than '='} && (5)\\
\text{(Term)}\ T &::= \text{(as in first-order logic)} && (6)\\
\text{(Goal Clause)}\ Q &::= \text{:- } B && (7)
\end{aligned}
$$

Fig. 2.1: Syntax of a subset of GHC

a non-atomic value can free the storage occupied by the value or reuse it for other purposes. In order to achieve recycling, however, the implementation must be able to judge whether each read operation is the final one on the current value of the variable. Since this is difficult in general, a heap is usually managed using runtime garbage collection.

However, suppose a compiler guarantees, by static analysis, that some variable is used only for one-to-one communication. Then the storage occupied by the value can be freed or recycled immediately after it is read. In a concurrent setting, it is usually difficult to identify the final read operation on a variable used for one-to-many communication, but good news about one-to-one communication is that a read operation is always final.

This paper is concerned with concurrent logic programming in which logical (or single-assignment) variables are used as communication channels, and proposes a theoretical framework, called *linearity analysis*, that distinguishes between one-to-one and one-to-many communcation. We are particularly interested in Moded Flat GHC, [8] a concurrent logic language with strong moding, because its mode system infers information flow of logical variables and simplifies linearity analysis.

We have found, from concurrent logic programs written so far, that most logical variables are used for one-to-one communication. [10] In particular, virtually all of the variables with complex protocols such as incomplete messages and streams of streams are one-to-one. This suggests that linearity analysis can provide fundamental information for optimizing memory management.

## 2.2 Concurrent Logic Languages and Linearity Analysis

GHC (Guarded Horn Clauses) is a concurrent logic language whose syntax is shown in Figure 2.1. For simplicity, we assume that program clauses contain no guard goals (i.e., conditions of rewriting specified between ':-' and '|'), but this restriction is not essential for the theoretical framework developed in this paper.

The operational semantics of GHC models the concurrent reduction of goals starting with an initial goal clause. Reduction of a current goal clause involves either of the following:

- rewriting of a non-unification goal to (zero or more) goals, possibly after observing a required substitution (*ask*), or
- execution of a unification goal, which may publish a substitution (*tell*).

We review one-step reduction from a goal clause $Q$ to $Q'$. For notational simplicity, we identify a goal clause with the multiset of body goals in the goal clause.

- *Reduction of a non-unification goal $g \in Q$ using a clause (renamed using fresh variables) "$h$ :- \| $B$"*: The synchronization rule of GHC tells that there must be a substitution $\theta$ such that $g = h\theta$, and $Q' = Q \setminus \{g\} \cup B\theta$, where '$\setminus$' and '$\cup$' are multiset difference and union, respectively.
- *Reduction of a unification goal $(t_1 = t_2) \in Q$*: $Q' = (Q \setminus (t_1 = t_2))\theta$, where $\theta$ is the most general unifier of $t_1$ and $t_2$. We assume that the program is well-moded, [8] in which case the unification does not fail except due to occur check.

In either case, reduction in general involves the rewriting of a variable (say $v$) to a term $t$ ($\neq v$). In the reduction of a non-unification goal, $v$ must be a variable in the (renamed) program clause, while in the reduction of a unification goal, $v$ must be a variable in the goal clause. When $v$ has more than one input occurrence (occurrence that is rewritten to $t$ as the result of reduction), or equivalently, when $v$ is used for one-to-many communication, the number of pointers from $Q'$ to $t$ is increased. (Throughout the paper, we assume that assignment of a structured value is done by sharing rather than by copying.) The purpose of linearity analysis is to statically analyze exactly *where* shared data structures may occur—in which predicates, in which arguments, and in which part of the data structures taken by those arguments.

A data structure that has not been referenced by a variable for one-to-many communication is never shared by two or more readers. A compiler can know exactly when it is read finally and becomes garbage, and generate object code that returns the structure to a free list or recycles it locally.

## 2.3 Terminology

**Definition.** We say that an occurrence of a variable is a *channel occurrence* if it is the leftmost occurrence in a clause head or an occurrence in a clause body.

A variable can be thought of as a communication channel for one-shot or repetitive communication (the most typical of repetitive communication is stream communication), and a channel occurrence can be thought of as an endpoint of a channel. The

```
:- module main.
qsort(Xs,Ys) :- true | qsort(Xs,Ys,[]).

qsort([],    Ys0,Ys ) :- true | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
   part(X,Xs,S,L), qsort(S,Ys0,Ys1), Ys1=[X|Ys2],
   qsort(L,Ys2,Ys3).
part(_,[],    S, L ) :- true | S=[], L=[].
part(A,[X|Xs],S0,L ) :- A>=X | S0=[X|S], part(A,Xs,S,L).
part(A,[X|Xs],S, L0) :- A< X | L0=[X|L], part(A,Xs,S,L).
```

Fig. 2.2: A quicksort program

condition 'leftmost' is rather arbitrary; the motivation is that only one of the (possibly many) occurrences of a variable can be called a channel occurrence. The condition does not imply that the arguments in a clause head are processed from left to right.

**Definition.** A variable that has at most two channel occurrences in a program clause or a goal clause is called a *linear* variable, and a variable that possibly has three or more occurrences is called a *nonlinear* variable.

Thus it is always safe to say some variable is nonlinear, but the purpose of linearity analysis is to detect as many linear variables as possible.

**Example.** In the quicksort program shown in Figure 2.2, all the variables except `X` in the second clause of ternary `qsort` are *linear*.

Strong moding guarantees that each variable generated during program execution has exactly one *output* occurrence, namely an occurrence that can determine its top-level value. This means that a variable with exactly two channel occurrences is used for one-to-one communication, and a variable with only one channel occurrence is used for one-to-zero communication.

**Definition.** A *path* is a sequence of pairs, of the form $\langle symbol, arg \rangle$, of function/predicate symbols and argument positions. In this paper, we regard constant symbols as nullary function symbols. Paths are used to specify occurrences of variables or function symbols in a goal or a term. Let $P_{Atom}$ be the set of all paths for specifying occurrences in goals, and $P_{Term}$ the set of all paths for specifying occurrences in terms.

For example, a function symbol `b` occurs in a goal `p(f(a,b),C)` at $\langle \mathtt{p}, 1 \rangle \langle \mathtt{f}, 2 \rangle \in P_{Atom}$. An empty sequence in $P_{Term}$ specifies the principal function symbol of a term in question.

## 2.4 Linearity Annotation

To distinguish between non-shared and shared data structures in a computational model without the notion of pointers, we consider giving a linearity annotation 1 or $\omega$ to every occurrence of a function symbol $f$ appearing in (initial or reduced) goal clauses and body goals in program clauses.[2] The annotations appear as $f^1$ or $f^\omega$ in the theoretical framework, though the purpose of linearity analysis is to reason about the annotations and compile them away so that the program can be executed without having to maintain linearity annotations at run time.

Intuitively, the principal function symbol of a structure possibly referenced by more than one pointer must have the annotation $\omega$, while a structure always pointed to by only one pointer in its lifetime can have the annotation 1. Another view of the annotation is that it models a one-bit reference counter that is not decremented once it reaches $\omega$.

The annotations must observe the following closure condition: If the principal function symbol of a term has the annotation $\omega$, all function symbols occuring in the term must have the annotation $\omega$. In contrast, a term with the principal function symbol annotated as 1 can contain a function symbol with either annotation, which means that a subterm of a non-shared term may possibly be shared.

Given linearity annotations, the operational semantics is extended to handle them so that they may remain consistent with the above intuitive meaning.

1. The annotations of function symbols in program clauses and *initial* goal clauses are given according to how the structures they represent are implemented. For instance, consider the following goal clause:

   ```
   :- p([1,2,3,4,5],X), q([1,2,3,4,5],Y).
   ```

   If the implementation chooses to create a single instance of the list `[1,2,3,4,5]` and let the two goals share them, the function symbols (there are 11 of them including `[]`) must be given $\omega$. If two instances of the list are created and given to `p` and `q`, either annotation is compatible with the implementation.

2. Suppose a substitution $\theta = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ is applied upon one-step reduction from $Q$ to $Q'$.

   (a) When $v_i$ is nonlinear, the substitution instantiates more than one occurrence of $v_i$ to $t_i$ and makes $t_i$ shared. Accordingly, all data structures inside $t_i$ (i.e., the subterms of $t_i$) become shared as well. So, prior to rewriting the occurrences of $v_i$ by $t_i$, we change all the annotations of the function symbols constituting $t_i$ to $\omega$.

   (b) When $v_i$ is linear, $\theta$ does not increase the number of references to $t_i$. So we rewrite $v_i$ by $t_i$ without changing the annotations in $t_i$.

[2] The notation is after related work [4, 7] on different computational models.

($\mathrm{BF}_\lambda$) If a function symbol $f^\omega$ occurs at the path $p$ in $B$, then
$\lambda(p) = \mathit{shared}$.

($\mathrm{LV}_\lambda$) If a linear variable occurs both at $p_1$ and $p_2$, then
$\forall q \in P_{\mathit{Term}}(m(p_1 q) = \mathit{in} \wedge \lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared})$
(if $p_1$ is a head path);
$\forall q \in P_{\mathit{Term}}(m(p_1 q) = \mathit{out} \wedge \lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared})$
(if $p_1$ is a body path).

($\mathrm{NV}_\lambda$) If a nonlinear variable occurs at $p$, then
$\forall q \in P_{\mathit{Term}}(m(pq) = \mathit{out} \Rightarrow \lambda(pq) = \mathit{shared})$ (if $p$ is a head path);
$\forall q \in P_{\mathit{Term}}(m(pq) = \mathit{in} \Rightarrow \lambda(pq) = \mathit{shared})$ (if $p$ is a body path).

($\mathrm{BU}_\lambda$) For a unification body goal $=_k$,
$\forall q \in P_{\mathit{Term}}(\lambda(\langle =_k, 1\rangle q) = \lambda(\langle =_k, 2\rangle q))$.

Fig. 2.3: Linearity constraints imposed by a clause $h$ :- | $B$

## 2.5 Linearity Constraints

The linearity of a well-moded program can be characterized using a linearity function.

**Definition.** A *linearity function* is a function from $P_{\mathit{Atom}}$ to the binary codomain $\{\mathit{nonshared}, \mathit{shared}\}$.

In this paper, we write $\lambda$ to stand for a linearity function.

The motivation of a linearity function is to distinguish between those paths where function symbols with $\omega$ *can* appear and those where function symbols with $\omega$ *cannot* appear. Suppose we can prove that a function symbol with $\omega$ cannot appear at $p$ such that $\lambda(p) = \mathit{nonshared}$. Then the (sole) reader of the data structure at a *nonshared* path can safely discard the top-level structure after accessing its elements. (There is one subtle point in this optimization, which will be discussed in Section 2.7.)

The above property can be established by enforcing *linearity constraints* on the function $\lambda$. Linearity constraints imposed by each program clause $h$ :- | $B$ or a goal clause :- $B$ are shown in Figure 2.3. The linearity constraints refer to the mode of a program represented by a function $m$. The mode constraints [8] on a well-moding $m$ are given in Figure 2.4. Here, a *submode* $m/p$ is defined as a function satisfying $(m/p)(q) = m(pq)$. The function $m/p$ represents the part of $m$ viewed at the path $p$. The functions $\mathit{IN}$ and $\mathit{OUT}$ are constant functions that always return *in* and *out*, respectively. An overline '$\overline{\ }$' inverts the polarity of a mode, a submode, or a mode value. We omit the motivations of each mode constraints and properties they enjoy. [8]

To allow different unification goals to have different modes and/or linearities, which is a limited form of polymorphism, each unification goal in program clauses and an initial

(HF) If a function symbol occurs at $p$ in $h$, then $m(p) = in$.

(HV) If a variable occurring $p$ in $h$ occurs elsewhere in $h$, then $m/p = IN$.

(BU) For a unification body goal $=_k$, $m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle}$.

(BF) If a function symbol occurs at $p$ in $B$, then $m(p) = in$.

(BV) Let a variable $v$ occur $n\,(\geq 1)$ times in $h$ and $B$ at $p_1, \ldots, p_n$, of which the occurrences in $h$ are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

$$\begin{cases} \mathcal{R}(\{m/p_1, \ldots, m/p_n\}), & k = 0; \\ \mathcal{R}(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}), & k > 0; \end{cases}$$

where $\mathcal{R}(S)$ is a 'cooperativeness' relation which states that, for all paths $q$, $\exists s \in S(s(q) = out \wedge \forall s' \in S \backslash \{s\}\,(s'(q) = in))$ holds.

Fig. 2.4: Mode constraints imposed by a clause $h$ :- | $B$

goal clause is given a unique serial number. In this paper, goals other than unification are assumed to be monomorphic; that is, different goals with the same predicate symbol have the same modes and linearities. This is for the sake of simplicity and it is possible to incorporate mode polymorphism [2] and in the same way linearity polymorphism.

The function $\lambda$ satisfying the linearity constraints is computed statically using program clauses and an initial goal clause. Linearity constraints in Figure 2.3 are trivially satisfied by letting $\lambda(p) = shared$ for all $p$. However, the purpose of linearity analysis is to compute the 'smallest' $\lambda$ satisfying linearity constraints, where the partial ordering is defined as

$$\lambda_1 \leq \lambda_2 \quad \Leftrightarrow \quad \forall p \in P_{Atom}(\lambda_1(p) = shared \Rightarrow \lambda_2(p) = shared).$$

How to solve linearity constraints to compute the smallest $\lambda$ will be discussed in Section 2.8.

## 2.6 Subject Reduction Theorem

This section gives a fundamental property that a linearity function enjoys.

**Definition.** Let $v$ be a variable and $t$ a term. We say that the *extended occur check* for unification between $v$ and $t$ *fails* if $t$ is $v$ or $t$ contains $v$.

**Theorem 1 (subject reduction).** Suppose $\lambda$ satisfies the linearity constraints of a program $P$ and a goal clause $Q$, and $Q$ is reduced in one step to $Q'$, where the reduced

goal $g \in Q$ is not a unification goal for which extended occur check fails. Then $\lambda$ satisfies the linearity constraints of $Q'$ as well.

**Proof.** Based on extensive case analysis. The cases can be divided into two based on whether the goal reduced is a non-unification goal or unification.

[Case 1] The reduction has rewritten a non-unification goal $g$ using a (renamed) clause $C \in P$ of the form "$h$ :- | $B$".

What we must consider are the constraints ($\mathrm{LV}_\lambda$) and ($\mathrm{NV}_\lambda$) imposed by the variables occurring in $g \in Q$ and the constraints ($\mathrm{BF}_\lambda$) imposed by the occurrences of function symbols brought into $B\theta$ ($\subseteq Q'$) by $\theta$ (these occurrences originate from the occurrences of the function symbols in $g$). Constraints imposed by the other variables in $Q'$ and those imposed by other occurrences of functions, which were already in either $Q \setminus \{g\}$ or $B$, need not be considered because they are exactly the same as those in $Q$. This means it suffices to consider all the symbols in $g$.

1. A function symbol $f^\kappa$ ($\kappa \in \{1, \omega\}$) occurs at the path $p$ in $g$. Then, either

   - the function $f$ occurs at $p$ in $h$, or
   - there exist $p' \in P_{Atom}$ and $q \in P_{Term}$ such that $p = p'q$ and a variable (say $v$) occurs at $p'$ in $h$.

   In the former case, ($\mathrm{BF}_\lambda$) is not applicable because the occurrence disappears upon reduction. So it suffices to consider the latter case, which may introduce new occurrences of $f$ into $B\theta$ ($\subseteq Q'$). Suppose $v$ occurs $n$ ($\geq 0$) times in $B$ at $r_1, \ldots, r_n$, and let $g_j$ be the goal to which the $j$th occurrence belongs. Then $f^{\kappa'}$ occurs in the goal $g_i\theta$ at $r_i q$, for $i = 1 \ldots, n$, where $\kappa' = \kappa$ if $n \leq 1$ and $\kappa' = \omega$ if $n > 1$ by Rule 2a in Section 2.4. We must show that $Q'$ enjoys ($\mathrm{BF}_\lambda$).

   (a) When $n \leq 1$ ($v$ is linear): It suffices to consider the case $n = 1$. If $\kappa = \omega$, $\lambda(r_i q) = \mathit{shared}$ must hold, but this can be derived as follows:

      i. $\lambda(p) = \mathit{shared}$, by ($\mathrm{BF}_\lambda$) applied to $f^\omega$ in $Q$,

      ii. $m(p) = \mathit{in}$, by (BF) applied to $f^\omega$ in $Q$,

      iii. $\lambda(p) = \mathit{shared} \Rightarrow \lambda(r_i q) = \mathit{shared}$, by 1(a)ii and ($\mathrm{LV}_\lambda$) applied to $v$ in $C$,

      iv. $\lambda(r_i q) = \mathit{shared}$, by 1(a)i and 1(a)iii.

   (b) When $n > 1$ ($v$ is nonlinear): Since $\kappa' = \omega$, we must show that $\lambda(r_i q) = \mathit{shared}$ holds.

      i. $m(p) = \mathit{in}$, by (BF) applied to $f^\kappa$ in $Q$,

      ii. $m(r_i q) = \mathit{in}$, by 1(b)i and (BV) applied to $v$ in $C$,

      iii. $\lambda(r_i q) = \mathit{shared}$, by 1(b)ii and ($\mathrm{NV}_\lambda$) applied to $v$ in $C$.

2. A variable $w$ occurs at $p$ in $g$. Suppose $w$ occurs $l$ ($\geq 1$) times in $g$ at $p_1(=p)$, $p_2, \ldots, p_l$ and $m$ ($\geq 0$) times in $Q\backslash\{g\}$ at $p_{l+1}, \ldots, p_{l+m}$.

   Because there exists $\theta$ such that $g = h\theta$, for each $p_i$ ($1 \leq i \leq l$), there exists a prefix $p'_i \in P_{Atom}$ of $p_i$ such that a variable (say $v_i$) occurs at $p'_i$ in $h$, and a path $q_i \in P_{Term}$ such that $p_i = p'_i q_i$. Suppose $v_i$ occurs $n_i$ ($\geq 0$) times in $B$ at $r_{i1}, \ldots, r_{in_i}$. Then $w$ is made to occur at $r_{ij}q_i$ ($1 \leq i \leq l$, $1 \leq j \leq n_i$) in $B\theta$.[3]

   When some $v_i$ is nonlinear, $w$ in $Q'$ becomes nonlinear as well. We consider those paths where $w$ occurs, namely

   - $r_{ij}q_i$ ($1 \leq i \leq l$, $1 \leq j \leq n_i$) (brought by $\theta$) and
   - $p_{l+1}, \ldots, p_{l+m}$ (inherited from $Q\backslash\{g\}$).

   (a) For the occurrences brought by $\theta$,

      i. $\forall q \in P_{Term}(m(r_{ij}q) = in \Rightarrow \lambda(r_{ij}q) = shared)$, by ($\mathrm{NV}_\lambda$) applied to $v_i$ in $C$,

      ii. $\forall q \in P_{Term}(m(r_{ij}q_iq) = in \Rightarrow \lambda(r_{ij}q_iq) = shared)$, by 2(a)i,

      so ($\mathrm{NV}_\lambda$) is satisfied for the paths of $w$ brought by $\theta$.

   (b) For the occurrences inherited from $Q \setminus \{g\}$, if $w$ is nonlinear in $Q$, ($\mathrm{NV}_\lambda$) applied to $Q'$ is immediate from ($\mathrm{NV}_\lambda$) applied to $Q$. If $w$ is linear in $Q$, we have $m \leq 1$ and now it suffices to consider the case where $m = 1$, namely the case where $w$ occurs at $p(= p_1)$ in $g$ and $p_2$ elsewhere. The goal is to show $\forall q \in P_{Term}(m(p_2q) = in \Rightarrow \lambda(p_2q) = shared)$, so we first assume $m(p_2q) = in$ for some $q$. Then

      i. $m(pq) = out$, by assumption and (BV) applied to $w$ in $Q$,

      ii. $\lambda(pq) = shared$, by 2(b)i and ($\mathrm{NV}_\lambda$) applied to $v_i$ in $C$,

      iii. $\lambda(p_2q) = shared$, by 2(b)i, 2(b)ii and ($\mathrm{LV}_\lambda$) applied to $w$ in $Q$.

      So ($\mathrm{NV}_\lambda$) is satisfied for the occurrences of $w$ inherited from $Q\backslash\{g\}$.

   When all the $v_i$'s are linear, $w$ is linear in $Q'$ if it is linear in $Q$, and nonlinear in $Q'$ otherwise. For each case, the linearity constraints to be satisfied by $Q'$ can be shown to hold with similar arguments.

[Case 2] The reduction has executed a unification goal $t_1 =_k t_2$. By the assumption of well-modedness, there exists an $i$ such that $m(\langle =_k, i\rangle) = out$. Without loss of generality, we can assume $i = 1$, in which case unification degenerates to assignment to the *left-hand side* variable. By the assumption of extended occur check, $t_2$ is not identical to the variable $t_1$ or a term containing $t_1$. So $Q'$ is equal to $(Q\backslash\{t_1 =_k t_2\})\{t_1 \leftarrow t_2\}$. Let the variable $t_1$ occur $n$ ($\geq 0$) times in $Q\backslash\{t_1 =_k t_2\}$ at $r_1, \ldots, r_n$. Then each symbol

[3] The variable $w$ may occur less than $n_1 + \cdots + n_l$ times because the two occurrences of $w$ in $g$ may be received by different occurrences of the same variable in $h$ and brought to $B\theta$ not independently.

in $t_2$ is duplicated $n$ times and occurs in $Q'$. It suffices to show that these occurrences enjoy the linearity constraints ($\mathrm{BF}_\lambda$), ($\mathrm{LV}_\lambda$), and ($\mathrm{NV}_\lambda$).

1. A function symbol $f^\kappa$ occurs at $\langle =_k, 2\rangle q$. The constraint ($\mathrm{BF}_\lambda$) tells that it suffices to consider the case $\kappa = \omega$.

   (a) $\lambda(\langle =_k, 2\rangle q) = \mathit{shared}$, by ($\mathrm{BF}_\lambda$) applied to $f^\omega$ in $Q$,

   (b) $\lambda(\langle =_k, 1\rangle q) = \mathit{shared}$, by 1a and ($\mathrm{BU}_\lambda$) applied to $Q$,

   (c) $m(\langle =_k, 2\rangle q) = \mathit{in}$, by (BF) applied to $f^\omega$ in $Q$,

   (d) $m(\langle =_k, 1\rangle q) = \mathit{out}$, by 1c and (BU) applied to $Q$,

   (e) $m(r_i q) = \mathit{in}$ $(1 \leq i \leq n)$, by 1d and (BV) applied to $t_1$ in $Q$,

   (f) when $t_1$ is linear, $\lambda(r_1 q) = \mathit{shared}$, by 1b, 1d and ($\mathrm{LV}_\lambda$) applied to $t_1$ in $Q$,

   (g) when $t_1$ is nonlinear, $\lambda(r_i q) = \mathit{shared}$ $(1 \leq i \leq n)$, by 1e and ($\mathrm{NV}_\lambda$) applied to $t_1$ in $Q$.

   By executing unification $t_1 =_k t_2$, $f^\omega$ is made to occur newly at $r_1 q, \ldots, r_n q$ in $Q'$. However, as shown above, $\lambda$ satisfies ($\mathrm{BF}_\lambda$) imposed by those new occurrences.

2. A variable $w$ $(\neq t_1)$ occurs at $\langle =_k, 2\rangle q$. Suppose $w$ occurs $l$ $(\geq 1)$ times in the goal $t_1 =_k t_2$ at $\langle =_k, 2\rangle q_1$, $\langle =_k, 2\rangle q_2$, $\ldots$, $\langle =_k, 2\rangle q_l$ and $m$ $(\geq 0)$ times in $Q \backslash \{t_1 =_k t_2\}$ at $p_{l+1}, \ldots, p_{l+m}$. By executing $t_1 =_k t_2$, $w$ is made to occur newly at $r_1 q_i, \ldots, r_n q_i$ $(1 \leq i \leq l)$. So it suffices to examine the linearity constraints of these paths.

   When $t_1$ is nonlinear in $Q$, $w$ in $Q'$ becomes nonlinear as well. However, by ($\mathrm{NV}_\lambda$) applied to $t_1$ in $Q$, $\forall s \in P_{\mathit{Term}}(m(r_j s) = \mathit{in} \Rightarrow \lambda(r_j s) = \mathit{shared})$ holds for $1 \leq j \leq n$, which implies ($\mathrm{NV}_\lambda$) applied to the new occurrences of $w$ in $Q'$.

   When $t_1$ and $w$ are both linear in $Q$, $w$ remains linear in $Q'$. We consider the less obvious case of $l = 2$ and $m = 0$, namely the case where the other occurrence of $w$ in $Q$ is also in $t_2$. (The other case where $l = 1$ and $m = 1$ is easier and thus omitted.) The goal is to show $\forall q \in P_{\mathit{Term}}(m(r_i q_i q) = \mathit{out} \wedge \lambda(r_i q_i q) = \mathit{shared} \Rightarrow \lambda(r_{3-i} q_{3-i} q) = \mathit{shared})$, for $i = 1, 2$. Without loss of generality we can focus on the case $i = 1$, so we first assume $m(r_1 q_1 q) = \mathit{out}$ and $\lambda(r_1 q_1 q) = \mathit{shared}$ for some $q$. Then

   (a) $\lambda(\langle =_k, 1\rangle q_1 q) = \mathit{shared}$, by ($\mathrm{LV}_\lambda$) applied to $t_1$ in $Q$,

   (b) $\lambda(\langle =_k, 2\rangle q_1 q) = \mathit{shared}$, by 2a and ($\mathrm{BU}_\lambda$),

   (c) $m(\langle =_k, 1\rangle q_i q) \neq m(r_i q_i q)$, by (BV) applied to $t_1$,

   (d) $m(\langle =_k, 1\rangle q_1 q) = \mathit{in}$, by the assumption and 2c,

   (e) $m(\langle =_k, 2\rangle q_1 q) = \mathit{out}$, by 2d and (BU),

   (f) $\lambda(\langle =_k, 2\rangle q_2 q) = \mathit{shared}$, by 2b, 2e and ($\mathrm{LV}_\lambda$) applied to $w$ in $Q$,

   (g) $\lambda(\langle =_k, 1\rangle q_2 q) = \mathit{shared}$, by 2f and ($\mathrm{BU}_\lambda$),

(h) $m(\langle =_k, 2\rangle q_2 q) = in$, by 2e and (BV) applied to $w$ in $Q$,

(i) $m(\langle =_k, 1\rangle q_2 q) = out$, by 2h and (BU),

(j) $\lambda(r_2 q_2 q) = shared$, by 2g, 2i, and ($\mathrm{LV}_\lambda$) applied to $t_1$ in $Q$.

When $t_1$ is linear in $Q$ and $w$ is nonlinear in $Q$, $w$ is in general nonlinear in $Q'$. In this case also, ($\mathrm{NV}_\lambda$) imposed by $w$ in $Q'$ can be derived in a similar manner from the linearity constraints of $Q$. Q.E.D.

Thus we have established that data structures occurring at a *nonshared* path in a goal in the course of computation are never shared.

## 2.7 Applications of Linearity Analysis

Linearity analysis provides fundamental information for the optimization of memory management that can potentially lead to novel applications of concurrent logic languages.

1. *Local reuse of data structures.* The sole reader of a data structure can recycle the structure it has read—for instance to create a new data structure. This enables update-in-place of data structures in a language without the notion of destructive assignments. Features like Lisp's `nconc` and `rplacd` need not be exposed to programmers any more.

   Local reuse may not necessarily have an impact on the performance of list processing on a single-processor machine, but it is essential in array processing in single-assignment languages. Despite their importance, arrays tend to be ignored in declarative languages. Since copying an array in each 'update' operation would be prohibitive, multi-version structures were often adopted as reasonable implementation of mutable arrays. However, if array variables are guaranteed to be linear, the implementation need not bother to create multi-version structures. Thus static linearity analysis seems essential to make declarative languages competitive with procedural languages in terms of performance. Linearity analysis enables not only update-in-place but also in-place splitting and merging of arrays. This opens up the possibility of *parallel* updating of a single array allocated on shared memory. [9]

   However, for concurrent logic programs, linearity analysis alone is not always sufficient for the local reuse of data structures due to the flexibility of logical variables. It *is* sufficient for the optimization of numeric or character arrays in which only instantiated data can be stored. When a data structure is allowed to contain uninstantiated logical variables and the writers of uninstantiated variables point directly to the (empty) slots of the data structure, the structure cannot be recycled until all the empty slots are filled and read. To enable local reuse in the

presence of partially instantiated data structures, analysis of instantiation states should be used together with linearity analysis.

2. *Distributed implementation.* In distributed applications in which pointers across sites can be limited to pointers to non-shared data, global garbage collection becomes unnecessary and the management of global pointers such as exporting and importing [5] can be greatly simplified. This opens up the possibility of using declarative languages in network programming applications in which program analysis and verification is still extremely difficult.

3. *Real-time and embedded applications.* In applications such as robot control, in which (soft) real-time processing is essential, an alternative to stop-and-copy garbage collection must be employed. A number of incremental and concurrent garbage collection algorithms have been proposed, [3] but compile-time garbage collection, where applicable, seems to be the most desirable solution to the problem. Linearity analysis is expected to play an important role in resource analysis as well—particularly the analysis of the amount of storage needed to execute a program. We believe that declarative programming with resource analysis will be a realistic tool for embedded and hard real-time applications.

## 2.8 Implementation—klint v2

A static analyzer for KL1 programs called *klint v2* [11] features both mode and linearity analyses. This section outlines the implementation of *klint v2.*

Basically, mode and linearity analyses are constraint satisfaction problems that can be solved using very similar techniques. In *klint v2,* a set of mode constraints is represented using a feature graph called a mode graph, [8] and solving a set of mode constraints means to merge (small feature graphs representing) new constraints into the 'current' mode graph, which is done mostly as unification over feature graphs. Non-binary constraints, which cannot be solved by unification, are imposed only by non-linear variables, and all the other constraints can be merged into the current mode graph within almost linear time with respect to the size of the mode graph. [8] For non-binary constraints, *klint v2* first postpones them in the hope that they become unary or binary by the information from other constraints. It turns out that many non-binary constraints are simplified finally.

When some constraints remain non-binary after solving all unary or binary constraints, *klint v2* assumes that nonlinear variables involved have simple, one-way dataflow rather than bidirectional dataflow such as in message streams with reply boxes. Thus, if a nonlinear variable occurs at $p$ and $m(p)$ is known to be *in* or *out*, *klint v2* imposes a stronger constraint $m/p = IN$ or $m/p = OUT$, respectively. This means that a mode graph computed by *klint v2* is not always most general, but the strengthening of constraints reduces most non-binary constraints to unary ones. Our observation is

that virtually all nonlinear variables have been used for one-way communication and the strengthening causes no problem in practice.

Following mode analysis, *klint v2* creates another feature graph called a *linearity graph*. Given the result of mode analysis, ($\mathrm{BF}_\lambda$) and ($\mathrm{NV}_\lambda$) are unary and ($\mathrm{BU}_\lambda$) is binary. However, ($\mathrm{LV}_\lambda$) is still a implicational constraint of the form $\lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared}$. Since most variables in a program are linear, it is unrealistic to implement an implicational constraint using delaying.

If the implication can be strengthened to a bidirectional one as in

- ($\mathrm{LV}'_\lambda$) $\forall q \in P_{\mathit{Term}}(\lambda(p_1 q) = \mathit{shared} \Leftrightarrow \lambda(p_2 q) = \mathit{shared})$,

the constraint can be solved using unification. Obviously ($\mathrm{LV}'_\lambda$) $\Rightarrow$ ($\mathrm{LV}_\lambda$) holds, and this approximation works well in detecting linear paths for most programs. However, consider a numeric array used as a shared look-up table. Such an array may well remain non-shared during initialization and then becomes shared. The change of the sharing property in the lifetime of a data structure is appropriately handled by ($\mathrm{LV}_\lambda$) using one-way constraint propagation, but with the approximated version ($\mathrm{LV}'_\lambda$), the structure is regarded as shared since its creation. This is undesirable because the initialization phase may very well want to exploit the efficiency of update-in-place.

*klint v2* circumvents this problem as follows. Since the data structures whose sharing property changes in their lifetime have simple dataflow (i.e., no bidirectional communication), we employ the full version ($\mathrm{LV}_\lambda$) only when $m/p_1$ and $m/p_2$ are known to be $\mathit{IN}$ or $\mathit{OUT}$, and the approximate version ($\mathrm{LV}'_\lambda$) otherwise. Suppose $p_1$ is a head path and $m/p_1$ is known to be $\mathit{IN}$. Then the first constraint of ($\mathrm{LV}_\lambda$) is simplified to

$$\forall q, r \in P_{\mathit{Term}}(\lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q r) = \mathit{shared}).$$

It turns out that this constraint is easy to implement using the notion of a propagator; that is, when $\lambda(p_1 q)$ is constrained to *shared*, it is propagated to the graph node representing $p_2$. A propagator is simply a graph edge (from $p_1$ to $p_2$) representing a 'null' feature, in contrast with other edges that represent $\langle \mathit{symbol}, \mathit{arg} \rangle$ features. A propagator is an essential tool for the eager evaluation of the constraint.

A graph node marked *shared* must express the closure condition $\forall p \in P_{\mathit{Atom}} \forall q \in P_{\mathit{Term}}(\lambda(p) = \mathit{shared} \Rightarrow \lambda(pq) = \mathit{shared})$, but this can be represented in much the same way as the representation of constant submode functions $\mathit{IN}$ and $\mathit{OUT}$.

As an example, we show the result of linearity analysis of the quicksort program shown in Figure 2.2 (Section 2.3).

```
*** Linearity Graph ***
node(0): (unconstrained)
 <(main:qsort)/2,1> ---> node(24)
 <(main:qsort)/2,2> ---> node(16)
 <(main:qsort)/3,1> ---> node(24)
```

```
 <(main:qsort)/3,2> ---> node(16)
 <(main:qsort)/3,3> ---> node(16)
 <(main:part)/4,1> ---> SHARED
 <(main:part)/4,2> ---> node(24)
 <(main:part)/4,3> ---> node(24)
 <(main:part)/4,4> ---> node(24)
node(24): (unconstrained)
 <cons,2> ---> node(24)
node(16): (unconstrained)
 <cons,1> ---> SHARED
 <cons,2> ---> node(16)
```

This is a textual representation of the linearity graph of quicksort. The paths indicated `SHARED`, namely

1. the first argument of `part`,
2. the elements of the list at the second argument of binary `qsort`, and
3. the elements of the lists at the second and the third arguments of ternary `qsort`

become shared no matter whether the input list from the first argument of binary `qsort` is non-shared or shared. However, all these paths are known to have scalar (integer) values by type analysis subsequently performed by *klint v2*. On the other hand, the list skeletons returned by the quicksort program is guaranteed to be non-shared.

## 2.9 Related Work

Study of the memory management of concurrent logic languages has a long history. A method that uses a one-bit reference counter called MRB (multiple reference bit) for each pointer was designed for Flat GHC [1] and adopted in a KL1 implementation on a Parallel Inference Machine. [5] Roughly speaking, linearity analysis proposed in this paper tries to compile away MRBs and related operations by analyzing the value of MRBs statically.

Janus [6] establishes the linearity property by allowing each variable to occur only twice. Our technique allows both linear and nonlinear variables and distinguishes between them by static analysis.

Various techniques for the distributed implementation of concurrent logic languages were proposed, [5] including import and export tables of pointers and weighted export counting. We are not claiming that all these techniques become unnecessary, but the management of data structures guaranteed to be non-shared by linearity analysis is greatly simplified.

Kobayashi proposes a type system with linearity information for the $\pi$-calculus. [4] In functional programming, Turner et al. introduce linearity annotation to the type system. [7] All these pieces of work could be considered the application of ideas with similar motivations to different computational models. In functional programming, the difficulty lies in the variety of evaluation rules and higher-order functions, while in concurrent logic programming, the difficulty lies in the treatment of arbitrarily complex information flow expressed by logical variables. Note that the mode and the linearity systems of Moded Flat GHC are essentially type systems in a broad sense.

## 2.10 Conclusions and Future Work

We have proposed a framework of linearity analysis for the concurrent logic language Moded Flat GHC and studied its fundamental property. Linearity analysis can be used with mode and type analyses to generate object code closer to that of procedural programs. Also, it opens up the possibility of writing distributed, embedded, and real-time software in a very simple concurrent programming language such as Moded Flat GHC and compiling them into safe and efficient code with systematic static analysis. Our future plan is to apply concurrent logic languages to the above areas where compilation into efficient code requires serious physical considerations.

## References


[1] Chikayama, T. and Kimura, Y., Multiple Reference Management in Flat GHC. In *Logic Programming: Proc. of the Fourth Int. Conf (ICLP'87)*, The MIT Press, 1987, pp. 276–293.

[2] Cho, K. and Ueda, K., Diagnosing Non-Well-Moded Concurrent Logic Programs. In *Proc. 1996 Joint Int. Conf. and Symp. on Logic Programming (JICSLP'96)*, The MIT Press, 1996, pp. 215–229.

[3] Jones, R. and Lins, R., *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Chichester, England, 1996.

[4] Kobayashi, N., Pierce, B. C. and Turner, D. N., Linearity and the Pi-calculus. In *Proc. 23rd ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL'96)*, ACM, 1996, pp. 358–371.

[5] Nakajima, K., Inamura, U., Ichiyoshi, N., Rokusawa, K. and Chikayama, T., Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. Sixth Int. Conf. on Logic Programming*, The MIT Press, 1989, pp. 436–451.

[6] Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conference on Logic*

*Programming*, Debray, S. and Hermenegildo, M. (eds.), The MIT Press, 1990, pp. 431–446.

[7] Turner, D. N., Wadler, P. and Mossin, C., Once Upon a Type. In *Proc. Seventh Int. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, ACM, 1995, pp. 1–11.

[8] Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.

[9] Ueda, K., Moded Flat GHC for Data-Parallel Programming. In *Proc. FGCS'94 Workshop on Parallel Logic Programming*, ICOT, Tokyo, 1994, pp. 27–35.

[10] Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems*, LNCS 1068, Springer, 1996, pp. 134–153.

[11] Ueda, K., *klint* — Static Analyzer for KL1 Programs. Available from `http://www.icot.or.jp/ARCHIVE/Museum/FUNDING/funding-98-E.html`, 1998.

# 第3章　資源交換の観点からみた並行計算とケイパビリティ型体系

The use of types to deal with access capabilities of program entities is becoming increasingly popular.

In concurrent logic programming, the first attempt was made in Moded Flat GHC in 1990, which gave polarity structures (modes) to every variable occurrence and every predicate argument. Strong moding turned out to play fundamental rôles in programming, implementation and the in-depth understanding of constraint-based concurrent computation.

The moding principle guarantees that each variable is written only once and encourages capability-conscious programming. Furthermore, it gives less generic modes to programs that discard or duplicate data, thus providing the view of "data as resources." A simple linearity system built upon the mode system distinguishes variables read only once from those read possibly many times, enabling compile-time garbage collection. Compared to linear types studied in other programming paradigms, the primary issue in constraint-based concurrency has been to deal with logical variables and highly non-strict data structures they induce.

In this paper, we put our resource-consciousness one step forward and consider a class of 'ecological' programs which recycle or return all the resources given to them while allowing concurrent reading of data structures via controlled aliasing. This completely recyclic subset enforces us to think more about resources, but the resulting programs enjoy high symmetry which we believe has more than aesthetic implications to our programming practice in general.

The type system supporting recyclic concurrent programming gives a $[-1, +1]$ capability to each occurrence of variable and function symbols (constructors), where positive/negative values mean read/write capabilities, respectively, and fractions mean non-exclusive read/write paths. The capabilities are intended to be statically checked or reconstructed so that one can tell the polarity and exclusiveness of each piece of information handled by concurrent processes. The capability type system refines and integrates the mode system and the linearity system for Moded Flat GHC. Its arithmetic formulation contributes to the simplicity.

The execution of a recyclic program proceeds so that every variable has zero-sum capability and the resources (i.e., constructors weighted by their capabilities) a process absorbs match the resources it emits. Constructors accessed by a process with an exclusive read capability can be reused for other purposes.

The first half of this paper is devoted to a tutorial introduction to constraint-based concurrency in the hope that it will encourage cross-fertilization of different concurrency formalisms.

## 3.1 Introduction – Constraint-Based Concurrency

The *raison d'être* and the challenge of symbolic languages are to construct highly sophisticated software which would be too complicated or unmanageable if written in other languages.

Concurrent logic programming was born in early 1980's from the process interpretation of logic programs [47] and forms one of many interesting subfields addressed by the logic programming paradigm [45].

The prominent feature of concurrent logic programming is that it exploits the power of logical, single-assignment variables and data structures – exactly those of first-order logic – to achieve various forms of communication.

Essentially, a logical variable is a communication channel that can be used for output at most once (hence single-assignment) and for non-destructive input zero or more times. The two well-established operations, unification and matching (also called one-way unification), are used for output and input. Thanks to the single-assignment property, the set of all unification operations that have been performed determines the current binding environment of the universe, which is called a (*monotonic*) *store* (of equality constraints) in concurrent constraint programming (CCP) [28] that generalizes concurrent logic programming. The store records what messages have been sent to what channels and what channels has been fused together.

In CCP, variable bindings are generalized to constraints, unification is generalized to the *tell* of a constraint to the store, and matching is generalized to the *ask* of a constraint from the store. The *ask* operation checks if the current store logically entails certain information on a variable.

Constraint-based communication embodied by concurrent logic programming languages has the following characteristics:

1. *Asynchronous.* In most concurrent logic languages, *tell* is an independent process that does not occur as a prefix of another process as in *ask*. This form of *tell* is sometimes called *eventual tell* and is a standard mechanism of information sending. (We do not discuss the other, prefixed form, *atomic tell*, in this paper.) Since *eventual tell* simply adds a new constraint to the store, the store can become inconsistent when an attempt is made to equate a variable to two different values. This can be avoided by using a non-standard type system, called a *mode system* [41], that controls the number of write capabilities of each variable in the system. The advocation of *eventual tell* apparently motivated Honda and Tokoro's asynchronous $\pi$-calculus [15].

2. *Polyadic.* Concurrent logic programming incorporated (rather than devised) well-understood built-in data structuring mechanisms and operations. Messages can be polyadic at no extra cost on the formalism; it does not bother us to encode numbers, tuples, lists, and so on, from scratch. The single-assignment property of logical variables does not allow one to use it for repetitive communication, but streams – which are just another name of lists in our setting – can readily be used for representing a sequence of messages incrementally sent from a process to another.

3. *Mobile.* A process[1] (say $P$) can dynamically create another process (say $P'$) and a fresh logical variable (say $v$) with which to communicate with $P'$. Although process themselves are not first-class, logical variables are first-class and its occurrences (other than the one 'plugged' to $P'$) can be freely passed from $P$ to other processes using another channel. The logical variable connected to a process acts as an object identity (or more precisely, channel identity because a process can respond to more than one channel) and the language construct does not allow a third process to forge the identity.

   When $P$ creates two occurrences of the variable $v$ and sends one of them to another process (say $Q$), $v$ becomes a private channel between $P'$ and $Q$ that cannot be monitored by any other process unless $P'$ or $Q$ passes it to somebody else. This corresponds to scope extrusion in the $\pi$-calculus.

   Another form of reconfiguration happens when a process fuses two logical variables connected to the outside. The fusion makes sense when one of the variables can be read (input capability) and the other can be written (output capability), in which case the effect of fusing is implicit delegation of messages. Again, the process that fused two logical variables loses access to them unless it retains a copy of them. As will be discussed later, our type systems have dealt with read/write capabilities of logical variables and the number of access paths (occurrences) of each variable.

   Although not as widely recognized as it used to be, Concurrent Prolog [30] designed in early 1980s was the first simple high-level language that featured channel mobility in the sense of the $\pi$-calculus. When the author proposed Guarded Horn Clauses (GHC) [36] [37] as a simplification of Concurrent Prolog and PARLOG [8], the principal design constraint was to retain channel mobility and evolving process structures [32], because GHC was supposed to be the basis of KL1 [39], a language in which to describe operating systems of the Parallel Inference Machines as well as various knowledge-based systems.

4. *Non-strict.* Logical variables provide us with the paradigm of *computing with partial information.* Interesting programming idioms including short-circuits, dif-

[1] We regard a process as an entity that is implemented as a multiset $S$ of goals and communicates with other processes by generating and observing constraints on variables not local to $S$.

| | | |
|---|---|---|
| (program) | $P$ ::= set of $R$'s | (1) |
| (program clause) | $R$ ::= $A$ :- \| $B$ | (2) |
| (body) | $B$ ::= multiset of $G$'s | (3) |
| (goal) | $G$ ::= $T_1 = T_2$ \| $A$ | (4) |
| (non-unification atom) | $A$ ::= $p(T_1, \ldots, T_n)$, $p \neq$ '=' | (5) |
| (term) | $T$ ::= (as in first-order logic) | (6) |
| (goal clause) | $Q$ ::= :- $B$ | (7) |
| (program clause, alternative) | $R$ ::= $!\forall(A \,.\, B)$ | (2′) |
| (goal clause, alternative) | $Q$ ::= $B, P$ | (7′) |

Fig. 3.1: The simplified syntax of Flat GHC

ference lists and messages with reply boxes, as well as channel mobility, all exploit the power of partially instantiated data structures.

Some historical remarks would be appropriate here.

Concurrent logic programming was a field of active research throughout the 1980's, when a number of concurrent logic languages were proposed and the language constructs were tested through a number of implementations and applications [31]. The synchronization primitive, now known as *ask* based on logical entailment, was inspired independently around 1984 by at least three research groups, which suggests the stability of the idea [32].

Although concurrent logic languages achieved their flexibility with an extremely small number of language constructs, the fact that they were targeted to programming rather than reasoning about concurrent systems lead to little cross-fertilization with later research on mobile processes.

CCP was proposed in late 1980s as a unified theory underlying concurrent logic languages. It helped high-level understanding of constraint-based concurrency, but the study of constraint-based communication at a concrete level and the design of type systems and static analyses call for a fixed constraint system – most typically that of (concurrent) logic programming known as the Herbrand system – to work with.

## 3.2 The Essence of Constraint-Based Communication

### 3.2.1 The Language

To further investigate constraint-based communication, let us consider a concrete language, a subset of Flat GHC [38] whose syntax is given in Fig. 3.1.

For simplicity, the syntax given in Fig. 3.1 omits guard goals from (2), which correspond to conditions in conditional rewrite rules. We use the traditional rule-based

$$\frac{\langle B_1, C, P\rangle \longrightarrow \langle B_1', C', P\rangle}{\langle B_1 \cup B_2, C, P\rangle \longrightarrow \langle B_1' \cup B_2, C', P\rangle} \quad \text{(i)}$$

$$\frac{}{\langle \{t_1 = t_2\}, C, P\rangle \longrightarrow \langle \emptyset, C \cup \{t_1 = t_2\}, P\rangle} \quad \text{(ii)}$$

$$\frac{}{\langle \{b\}, C, \{h :- \mid B\} \cup P\rangle \longrightarrow \langle B, C \cup \{b = h\}, \{h :- \mid B\} \cup P\rangle} \begin{pmatrix} \mathcal{E} \models \forall(C \Rightarrow \exists \mathcal{V}_h(b = h)) \\ \text{and } \mathcal{V}_{h,B} \cap \mathcal{V}_{b,C} = \emptyset \end{pmatrix} \quad \text{(iii)}$$

Fig. 3.2: The reduction semantics of GHC

syntax rather than the expression-based one because it facilitates our analysis. The alternative syntax (2′) (7′) indicates that a program clause, namely a rewrite rule of goals, could be regarded as a replicated process that accepts a message $A$ and spawns $B$, where the universal closure $\forall$ means that a variable either occurs in $A$ and will be connected to the outside or occurs only in $B$ as local channels. In this formulation, the program is made to reside in a goal clause.

### 3.2.2 Operational Semantics

The reduction semantics of GHC deals with the rewriting of goal clauses.

A *configuration* is a triple, $\langle B, C, P\rangle$, where $B$ is a multiset of goals, $C$ a multiset of equations (denoting equality constraints) that represents the store, and $P$ a set of program clauses. A computation under a program $P$ starts with the initial configuration $\langle B_0, \emptyset, P\rangle$, where $B_0$ is the body of the given goal clause.

We have three rules given in Fig. 3.2. In the rules, $F \models G$ means that $G$ is a logical consequence of $F$. $\mathcal{V}_F$ denotes the set of all variables occurring in a syntactic entity $F$. $\forall \mathcal{V}_F(F)$ and $\exists \mathcal{V}_F(F)$ are abbreviated to $\forall(F)$ and $\exists(F)$, respectively. $\mathcal{E}$ denotes the standard syntactic equality theory over finite terms and atomic formulas defined in Fig. 3.3. The second condition of Fig. 3.3, characterizing the finiteness of terms, is known as the *occur check*.

In Fig. 3.2, Rule (i) expresses concurrent reduction of a multiset of goals. Rule (ii) says that a unification goal simply publishes (or posts) a constraint to the current store. Rule (iii) deals with the reduction of a non-unification goal $b$ to $B$ using a clause $h :- \mid B$, which is enabled when the publication of $b = h$ will not constrain the variables in $b$. This means that the head unification is effectively restricted to matching. The second side condition guarantees that the guarded clause has been renamed using fresh variables. An immediate consequence of Rules (i)–(iii) is that the store grows monotonically and the reduction of $b$ using a clause $h :- \mid B$ remains enabled once it becomes enabled.

1. $\forall(\neg(f(\mathtt{X}_1,\ldots,\mathtt{X}_m)=g(\mathtt{Y}_1,\ldots,\mathtt{Y}_n)))$, for all pairs $f$, $g$ of distinct constructors (including constants)
2. $\forall(\neg(t=\mathtt{X}))$, for each term $t$ other than and containing $\mathtt{X}$
3. $\forall(\mathtt{X}=\mathtt{X})$
4. $\forall(f(\mathtt{X}_1,\ldots,\mathtt{X}_m)=f(\mathtt{Y}_1,\ldots,\mathtt{Y}_m)\Rightarrow\bigwedge_{i=1}^{m}(\mathtt{X}_i=\mathtt{Y}_i))$, for each $m$-ary constructor $f$
5. $\forall(\bigwedge_{i=1}^{m}(\mathtt{X}_i=\mathtt{Y}_i)\Rightarrow f(\mathtt{X}_1,\ldots,\mathtt{X}_m)=f(\mathtt{Y}_1,\ldots,\mathtt{Y}_m))$, for each $m$-ary constructor $f$
6. $\forall(\mathtt{X}=\mathtt{Y}\Rightarrow\mathtt{Y}=\mathtt{X})$
7. $\forall(\mathtt{X}=\mathtt{Y}\wedge\mathtt{Y}=\mathtt{Z}\Rightarrow\mathtt{X}=\mathtt{Z})$

Fig. 3.3: Clark's equality theory $\mathcal{E}$, in clausal form

Sometimes it's more convenient to treat reduction in a traditional way as rewriting of goal clauses. The goal clause corresponding to a configuration $\langle B, C, P\rangle$ is $:- B\theta$, where $\theta$ is the most general unifier (mgu) of the set $C$ of constraints. This substitution-based formulation is closer to actual implementation, but an advantage of the constraint-based formulation is that it can represent inconsistent stores, while mgu's can represent consistent stores only.

Yet another formulation may omit the second component, $C$, of a configuration together with Rule (ii) that simply moves an unguarded unification goal to the separate store. In this case, the current store is understood to comprise all the unguarded unification goals in $B$. However, we think it makes sense to distinguish between the three entities, namely definitions (code), processes, and the store.

### 3.2.3 Relation to Name-Based Concurrency

How can the constraint-based concurrency defined above relates to name-based concurrency?

First of all, predicate names can be thought of as global channel names if we regard the reduction of a non-unification goal (predicate name followed by arguments) as message sending to predicate definition. However, we don't regard this as a crucially important observation. We would rather forget this correspondence and focus on other symbols, namely variables and constructors.

Variables are local names that can be used as communication channels. Instead of *sending* a message *along* a channel, the the message is written to the channel itself and the receiver can asynchronously read the channel's value. For instance, let `S` be shared by processes $P$ and $Q$ (but nobody else) and suppose $P$ sends a message `S=[read(X)|S']`. The message sends two subchannels, one a reply box `X` for the request

`read`, and the other a *continuation* for subsequent communication. Then the goal in $Q$ that owns `S`, say `q(S)`, can read the message using a clause head `q([read(A)|B])`, identifying `A` with `X` and `B` with `S'` at the same time.[2] Alternatively, the identification of variables can be dispensed with by appropriately choosing an $\alpha$-converted variant of the clause.

There is rather small difference between message passing of the asynchronous $\pi$-calculus and message passing by unification, as long as only one process holds a write capability *and* use it once. These conditions can be statically checked in well-moded concurrent logic programs [41] and in the $\pi$-calculus with a linear type system [19]. When two processes communicate repeatedly, constraint-based concurrency uses streams because one fresh logical variable must be prepared for each message passing, while in the linear $\pi$-calculus the same channel could be recycled as suggested in [19]. When two client processes communicate with a single server in constraint-based concurrency, an arbitration process should be explicitly created. A stream merger is a typical arbiter for repetitive multi-client communication:

```
merge([],Ys,Zs) :- | Zs=Ys.
merge(Xs,[],Zs) :- | Zs=Xs.
merge([A|Xs],Ys,Zs0) :- | Zs0=[A|Zs], merge(Xs,Ys,Zs).
merge(Xs,[A|Ys],Zs0) :- | Zs0=[A|Zs], merge(Xs,Ys,Zs).
```

In contrast, in name-based concurrency (without linearity), arbitration is built in the communication rule

$$a(y).Q \mid \overline{a}b \longrightarrow Q\{b/y\}$$

which chooses one of available outputs (forming a multiset of messages) on the channel $a$.

The difference in the semantics of input is much larger between the two formalisms. While *ask* is an non-destructive input, input in name-based concurrency destructively consumes a message, which is one of the sources of nondeterminism in the absence of choice operators. In constraint-based concurrency, non-destructiveness of *ask* is used to model one-way multicasting or data sharing naturally. At the same time, by using a linearity system, we can guarantee that only one process holds a read capability of a logical variable [46], in which case *ask* can destroy a message it has received, as will be discussed in detail in this paper.

One feature of constraint-based concurrency included into name-based concurrency only recently by the Fusion calculus [48] is that two channels can be fused into a single channel.

[2] In the syntax advocated by CCP, one should first ask $\exists$`A,B(q(S)=q([read(A)|B]))` (or equivalently, $\exists$`A,B(S=[read(A)|B]))` first and then tell `q(S)=q([read(A)|B])`.

### 3.2.4 Locality in Global Store

The notion of shared, global store provided by CCP must be understood with care. Unlike conventional shared-memory multiprocessing, constraint store of CCP is highly structured and localized. All channels in constraint-based concurrency are created as local variables most of which are shared by two or a small community of processes, and a process can access them only when they are explicitly passed as (part of) messages or by fusing.

The only names understood globally are

1. predicate symbols used as the names of recursive programs, and
2. function symbols (constructors) for composing messages, streams, and data structures, and so on.

Although predicate symbols could be considered as channels, they are channels to classes rather than to objects. Constructors are best considered as non-channel names. They have various rôles as above, but cannot be used for sending messages through them. They can be examined by matching (*ask*) but cannot be equated with other constructors under strong moding.

## 3.3 I/O Mode Analysis

### 3.3.1 Motivation

By early 1990's, hundreds of thousands of lines of GHC/KL1 code were written inside and outside the Fifth Generation Computer Project [32]. The applications include an operating system for the Parallel Inference Machine (PIMOS) [6], a parallel theorem prover (MGTP) that discovered a new fact in finite algebra [12]. genetic information processing, and so on.

People found the communication and synchronization mechanisms of GHC/KL1 very natural. Bugs due to concurrency were rather infrequent[3] and people learned to model their problems in an object-based manner using concurrent processes and streams. At the same time, writing efficient *parallel* programs turned out to be a separate and much harder issue than writing correct *concurrent* programs.

By late 1980's, we had found that logical variables in concurrent logic languages were normally used for cooperative rather than competitive communication. Because the language and the model based on eventual *tell* provided no mechanism to cope with the inconsistency of a store (except for exception handers of KL1) and an inconsistent store allows any constraint to be read out, it was the responsibility of the programmers to keep the store consistent. Although shared logical variables were sometimes used

[3]Most bugs were due to higher-level design problems that often arose in, for example, programs dealing with circular process structures concurrently.

for $n$-to-$n$ signalling, in which two or more processes could write the same value to the same variable, for most applications it seemed desirable to provide syntactic control of interference so that the consistency of the store could be guaranteed statically. Obviously, a store remains consistent if only one process is allowed to have a write capability of each variable, as long as we ignore the *occur check* condition (Sect. 3.2.2).

The mode system[4] of Moded Flat GHC [41][43] was designed to establish this property while retaining the flexibility of constraint-based communication as much as possible. Furthermore, we can benefit very much from strong moding, as we do from strong typing in many other languages:

1. It helps programmers understand their programs better.
2. It detects a certain kind of program errors at compile-time. In fact, the Kima system we have developed [2][3] goes two steps forward: it *locates*, and then automatically *corrects*, simple program errors using constraint-based mode and type analyses. The technique used in Kima is very general and could be deployed in other typed languages as well.
3. It establishes some fundamental properties statically (Sect. 3.3.5):
   (a) well-moded programs do not collapse the store.
   (b) all variables are guaranteed to become *ground* terms upon termination.
4. It provides basic information for program optimization such as
   (a) elimination of various runtime checks,
   (b) (much) simpler distributed unification, and
   (c) message-oriented implementation [41][40].

### 3.3.2 The Mode System

The purpose of our mode system is to assign *polarity structures* (modes) to every predicate argument and (accordingly) every variable occurrence in a configuration, so that each part of data structures will be determined cooperatively, namely by *exactly one* process that owned a write capability. If more than one process owned a write capability to determine some part a structure, the communication would be competitive rather than cooperative. If no process owned a write capability, the communication would be neither cooperative or competitive, because the readers would never get a value.

Since variables may be bound to complex data structures in the course of computation whose exact shapes are not known beforehand, a polarity structure reconstructed by

[4]Modes have sometimes been called directional types. In any case modes are (non-standard) types that deal with read/write capabilities.

the mode system should tell the polarity structures of all possible data structures the program may create and read. To this end, a mode is defined as a function from the set of paths specifying positions in data structures occurring in goals, denoted $P_{Atom}$, to the set $\{in, out\}$. Paths here are strings of $\langle$*symbol, argument-position*$\rangle$ pairs in order to be able to specify positions in data structures that are yet to be formed.

Formally, the sets of paths for specifying positions in terms and atomic formulas are defined, respectively, using disjoint union as:

$$P_{Term} = (\sum_{f \in Fun} N_f)^* \ , \quad P_{Atom} = (\sum_{p \in Pred} N_p) \times P_{Term} \ ,$$

where *Fun* and *Pred* are the sets of constructors and predicate symbols, respectively, and $N_f$ and $N_p$ are the sets of positive integers up to and including the arities of $f$ and $p$, respectively.

### 3.3.3 Mode Analysis

Mode analysis tries to find a mode $m : P_{Atom} \rightarrow \{in, out\}$ under which every piece of communication will be performed cooperatively. Such a mode is called a *well-moding*. A well-moding is computed by constraint solving. Constructors in a program/goal clause will impose constraints on the possible polarities of the paths at which they occur. Variable symbols may constrain the polarities not only of the paths at which they occur but of any positions below those paths. The set of all these mode constraints syntactically imposed by the symbols or the symbol occurrences in a program does not necessarily define a unique mode because the constraints are usually not strong enough to define one. Instead it defines a 'principal' mode that can best be expressed as a mode graph, as we will see in Section 3.3.6.

Mode constraints imposed by a clause $h$ :- | $B$, where $B$ are multisets of atomic formulae, are summarized in Fig. 3.4. Here, *Var* denotes the set of variable symbols, and $\widetilde{a}(p)$ denotes a symbol occurring at $p$ in an atomic formula $a$. When $p$ does not lead to a symbol in $a$, $\widetilde{a}(p)$ returns $\perp$. A *submode* of $m$ at $p$, denoted $m/p$, is a function (from $P_{Term}$ to $\{in, out\}$) such that $(m/p)(q) = m(pq)$. *IN* and *OUT* are constant submodes that always return *in* or *out*, respectively. An overline, "$-$", inverts the polarity of a mode, a submode, or a mode value.

For goal clauses, Rules (BU), (BF) and (BV) are applicable.

Note that Rule (BV) ignores the second and the subsequent occurrences of $v$ in $h$. The occurrences of $v$ that are not ignored are called *channel occurrences*. Note also that $s$ can depend on $q$ in the definition of $\mathcal{R}$. Intuitively, Rule (BV) means that each constructor occurring in a possible instance of $v$ will be determined by exactly one of the channel occurrences of $v$.

Unification body goals, dealt with by Rule (BU), are polymorphic in the sense that different goals are allowed to have different modes. To deal with polymorphism, we

(HF) $\forall p \in P_{Atom}(\widetilde{h}(p) \in Fun \Rightarrow m(p) = in)$
(if the symbol at $p$ in $h$ is a constructor, $m(p) = in$)

(HV) $\forall p \in P_{Atom}(\widetilde{h}(p) \in Var \wedge \exists p' \neq p(\widetilde{h}(p) = \widetilde{h}(p')) \Rightarrow m/p = IN)$
(if the symbol at $p$ in $h$ is a variable occurring elsewhere in $h$, then $m/p = IN$)

(BU) $\forall k > 0 \forall t_1, t_2 \in Term((t_1 =_k t_2) \in B \Rightarrow m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle})$
(the two arguments of a unification body goal have complementary submodes)

(BF) $\forall p \in P_{Atom} \forall a \in B(\widetilde{a}(p) \in Fun \Rightarrow m(p) = in)$
(if the symbol at $p$ in a body goal is a constructor, $m(p) = in$)

(BV) Let $v \in Var$ occur $n\ (\geq 1)$ times in $h$ and $B$ at $p_1, \ldots, p_n$, of which the occurrences in $h$ are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

$$\begin{cases} \mathcal{R}(\{m/p_1, \ldots, m/p_n\}), & k = 0; \\ \mathcal{R}(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}), & k > 0; \end{cases}$$

where $\mathcal{R}$ is a 'cooperativeness' relation:

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \forall q \in P_{Term}\, \exists s \in S(s(q) = out \wedge \forall s' \in S \backslash \{s\}\, (s'(q) = in))$$

Fig. 3.4: Mode constraints imposed by a clause $h$ :- | $B$

give each unification body goal a unique number. Polymorphism can be incorporated to other predicates as well [43], but we do not discuss it here.

### 3.3.4 Moding Principles

What are the principles behind these moding rules?

In concurrent logic programming, a process implemented by a multiset of goals can be considered an information processing device with inlets and outlets of constraints that we call *terminals*. A variable is a one-to-$n$ $(n \geq 0)$ communication channel connecting its occurrences, and each occurrence of a variable is considered to be plugged into one of the terminals of a goal.

We say that a variable is *linear* when it has exactly two occurrences in a goal clause. Similarly, a variable in a program clause is said to be *linear* when it has exactly two channel occurrences in the clause.

A variable occurring both in the head and in the body of a program clause is considered a channel that connects a goal (which the head matches) and its subgoals. A constructor is considered an unconnected plug that acts as the source or the absorber of atomic information, depending on whether it occurs in the body or the head. While channels and terminals of electric devices usually have array structures, those in our

setting have nested structures. That is, a variable that connects the terminals at $p_1$, ..., $p_n$ also connects the terminals at $p_1 q$, ..., $p_n q$, for all $q \in P_{Term}$. Linear variables are used as *cables* for one-to-one communication, while nonlinear variables are used as *hubs* for one-to-many communication.

A terminal of a goal always has its counterpart. The counterpart of a terminal at $p$ on the caller side of a non-unification goal is the one at the same path on the callee side, and the counterpart of a terminal at $\langle =_k, 1 \rangle q$ in the first argument of a unification goal is the one at $\langle =_k, 2 \rangle q$ in the second argument. Reduction of a goal is considered the removal of the pairs of corresponding terminals whose connection has been established.

The mode constraints are concerned with the direction of information flow (1) in channels and (2) at terminals. The two underlying principles are:

1. When a channel connects $n$ terminals of which at most one is in the head, exactly one of the terminals is the outlet of information and the others are inlets.

2. Of the two corresponding terminals of a goal, exactly one is the outlet of information and the other is an inlet.

Rule (BV) comes from Principle 1. An input (output) occurrence of a variable in the head of a clause is considered an outlet (inlet) of information from inside the clause, respectively, and this is why we invert the mode of the clause head in Rule (BV). Rule (BV) takes into account only one of the occurrences of $v$ in the head. Multiple occurrences of the same variable in the head are for equality checking before reduction, and the only thing that matters after reduction is whether the variable occurs also in the body and conveys information to the body goals.

Rules (HF) and (HV) come from Principle 2. When some clause may examine the value of the path $p$ in a non-unification goal, $m(p)$ should be constrained to *in* because the examination is done at the *outlet* of information on the *callee* side of a goal. The strong constraint imposed by Rule (HV) is due to the semantics of Flat GHC: when a variable occurs twice or more in a clause head, these occurrences must receive identical terms from the caller.

Rule (BU) is exactly the application of Principle 2 to unification body goals. Any value fed through some path $\langle =_k, i \rangle q$ in one of its arguments will come out through the corresponding path $\langle =_k, 3 - i \rangle q$ in the other argument.

Rule (BF) also comes from Principle 2. A non-variable symbol on the caller side of a goal must appear only at the inlet of information, because the information will go out from the corresponding outlet.

The relation $\mathcal{R}$ enjoys the following properties:

$$\mathcal{R}(\{s\}) \Leftrightarrow s = OUT \tag{3.1}$$

$$\mathcal{R}(\{s_1, s_2\}) \Leftrightarrow s_1 = \overline{s_2} \tag{3.2}$$

$$\mathcal{R}(\{IN\} \cup S) \Leftrightarrow \mathcal{R}(S) \tag{3.3}$$

$$\mathcal{R}(\{OUT\} \cup S) \Leftrightarrow \forall s' \in S(s' = IN) \quad (3.4)$$

$$\mathcal{R}(\{s, s\} \cup S) \Leftrightarrow s = IN \wedge \mathcal{R}(S) \quad (3.5)$$

$$\mathcal{R}(\{\overline{s}, s\} \cup S) \Leftrightarrow \forall s' \in S(s' = IN) \quad (3.6)$$

$$\mathcal{R}(\{\overline{s}\} \cup S_1) \wedge \mathcal{R}(\{s\} \cup S_2) \Rightarrow \mathcal{R}(S_1 \cup S_2) \quad (3.7)$$

$$\mathcal{R}(\bigcup_{1 \le i \le n} \{s_i\}) \Rightarrow \mathcal{R}(\bigcup_{1 \le i \le n} \{s_i/q\}), \quad q \in P_{Term} \quad (3.8)$$

Proofs are all straightforward. Property (3.7) is reminiscent of Robinson's resolution principle.

Properties (3.1) and (3.2) say that Rule (BV) becomes much simpler when the variable $v$ has at most two channel occurrences. When it has exactly two channel occurrences at $p_1$ and $p_2$. Rule (BV) is equivalent to $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$, depending on whether one of the occurrences is in the head or the both occur in the body. When $v$ has only one channel occurrence at $p$, Rule (BV) is equivalent to $m/p = IN$ or $m/p = OUT$, depending on whether the occurrence is in the head or the body.

### 3.3.5 Properties of Well-Moded Programs

The three important properties of well-moding are as follows:

1. Let $m$ be a well-moding of a clause $R$, and let $t_1 =_k t_2$ be a unification (body) goal in $R$. Then there exists an $i$ such that (i) $m(\langle =_k, i \rangle) = out$ and (ii) $t_i$ is a variable.

   This means a unification body goal is effectively assignment to an variable with a write capability.

2. (*Subject Reduction*) Let $m$ be a well-moding of a program $P$ and a goal clause $Q$. Suppose $Q$ is reduced by one step into a goal clause $Q'$ (in the substitution-based formulation (Sect. 3.2.2)), where the reduced goal $g \in Q$ is *not* a unification goal that unifies a variable with itself or a term containing the variable. Then $m$ is a well-moding of $P$ and $Q'$ as well.

   As a corollary, well-moded programs keep store consistent as long as the reductions obey the above condition on the reduced goal, which is called the *extended occur-check condition.*

3. (*Groundness*) Let $m$ be a well-moding of a program $P$ and a goal clause $Q$. Assume $Q$ has been reduced to an empty multiset of goals under the extended occur-check condition. Then, in that execution, a unification goal of the form $v =_k t$ such that $m(\langle =_k, 1 \rangle) = out$, or a unification goal of the form $t =_k v$ such that $m(\langle =_k, 2 \rangle) = out$, must have been executed, for any variable $v$ occurring in $Q$.

   As a corollary, the product of all substitutions generated by unification body goals maps all the variables in $Q$ to ground (variable-free) terms.

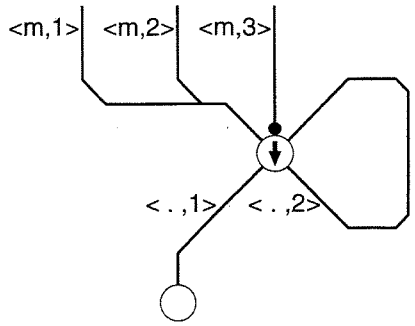



Fig. 3.5: Mode graph of the `merge` program

### 3.3.6 Mode Graphs and Principal Modes

It turns out that most of the mode constraints are either of the six forms: (i) $m(p) = in$, (ii) $m(p) = out$, (iii) $m/p = IN$, (iv) $m/p = OUT$, (v) $m/p_1 = m/p_2$, or (vi) $m/p_1 = \overline{m/p_2}$. We call (i)–(iv) *unary* constraints and (v)–(vi) *binary* constraints.

A set of binary and unary mode constraints can be represented as a feature graph (feature structures with cycles), called a *mode graph*, in which

1. paths represent paths in $P_{Atom}$,
2. nodes may have mode values determined by unary constraints,
3. arcs may have "negative signs" that invert the interpretation of the mode values beyond those arcs, and
4. binary constraints are represented by the sharing of nodes.

Figure 3.5 is the mode graph of the `merge` program. An arc of a mode graph represents the pair of a predicate/constructor (abbreviated to its initial in the figures) and an argument position. A dot "." stands for the list constructor. The pair exactly corresponds to a feature of a feature graph. A sequence of features forms a path both in the sense of our mode system and in the graph-theoretic sense.

A node is possibly labeled with a mode value (*in* shown as "↓", or *out* shown as "↑") to which any paths $p_1$, $p_2$, ... terminating with that node are constrained, or with a constant submode (*IN* shown as "↓" with a grounding sign (as in Fig. 3.7), or *OUT*) to which the submodes $m/p_1$, $m/p_2$, ... are constrained.

An arc is either a negative arc (bulleted in the figures) or a positive arc. When a path passes an odd number of negative arcs, that path is said to be *inverted*, and the mode value of the path is understood to be inverted. Thus the number of bulleted arcs on a path determines the *polarity* of the path.

A binary constraint of the form $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$ is represented by a shared node with two (or more) incoming paths with possibly different polarities.

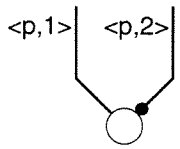



Fig. 3.6: Mode graph of the unify program

When the polarities of the two incoming paths are different, the shared node stands for complementary submodes; otherwise the node stands for identical submodes.

Figure 3.5 has a node, under the arc labeled $\langle ., 1\rangle$, that expresses no constraints at all. It was created to express binary constraints, but all its parent nodes were later merged into a single node by other constraints.

All these ideas have been implemented in the mode analyzer for KL1 program, *klint v2* [44], which can output a text version of the mode graph as in Fig. 3.5.

As another example, consider a program that simply unifies its arguments:

```
p(X,Y) :- | X = Y.
```

The program forms a mode graph shown in Fig. 3.6. This graph can be viewed as the *principal mode* of the predicate p, which represents many possible particular modes satisfying the constraint $m/\langle \mathrm{p}, 1\rangle = \overline{m/\langle \mathrm{p}, 2\rangle}$. In general, the principal mode of a well-moded program, represented as a mode graph, is uniquely determined, as long as all the mode constraints imposed by the program are unary or binary.

Constraints imposed by the rule (BV) may be non-binary. Non-binary constraints are imposed by nonlinear variables, and cannot be represented as mode graphs by themselves. However, by *delaying* them, most of them are reduced to unary/binary ones by other constraints. In this case they can be represented in mode graphs, and the programs that imposed them have unique principal modes (as long as they are well-moded). Theoretically, some non-binary constraints may remain unreduced, whose satisfiability must be checked eventually.

When some constraints remain non-binary after solving all unary or binary constraints, *klint v2* assumes that nonlinear variables involved are used for one-way multicasting rather than bidirectional communication. Thus, if a nonlinear variable occurs at $p$ and $m(p)$ is known to be *in* or *out*, *klint v2* imposes a stronger constraint $m/p = IN$ or $m/p = OUT$, respectively. This means that a mode graph computed by *klint v2* is not always 'principal', but the strengthening of constraints reduces most non-binary constraints to unary ones. Our observation is that virtually all nonlinear variables have been used for one-way multicasting and the strengthening causes no problem in practice.

The union (i.e., conjunction) of two sets of constraints can be computed efficiently as unification over feature graphs. For instance, adding a new constraint $m/p_1 = m/p_2$ causes the subgraph rooted at $p_1$ and the subgraph rooted at $p_2$ to be unified. A good

```
qsort([],     Ys0,Ys ) :- | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3) :- |
    part(X,Xs,S,L), qsort(S,Ys0,[X|Ys2]), qsort(L,Ys2,Ys3).
part(_,[],     S, L ) :- | S=[], L=[].
part(A,[X|Xs],S0,L ) :- A>=X | S0=[X|S], part(A,Xs,S,L).
part(A,[X|Xs],S, L0) :- A< X | L0=[X|L], part(A,Xs,S,L).
```

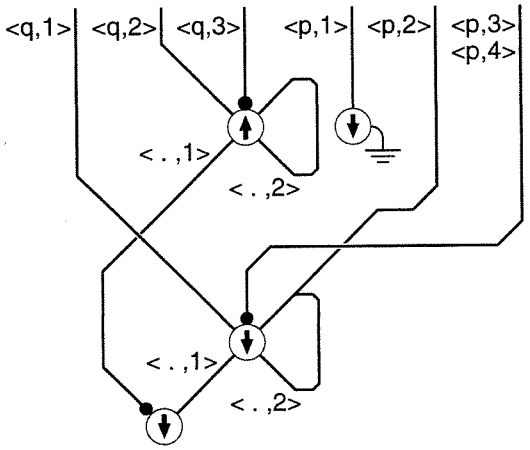



Fig. 3.7: A quicksort program and its mode graph

news is that an efficient unification algorithm for feature graphs has been established [1].

Figure 3.7 shows the mode graph of a quicksort program using difference lists. The second and the third clause of `part` checks the principal constructor of `A` and `X` using guard goals, so the moding rule of variables occurring in guard goals (not stated in this paper) constrains $m(\langle \texttt{part}, 1\rangle)$ and $m(\langle \texttt{part}, 2\rangle\langle ., 1\rangle)$ to *in*. The head and the tail of a difference list, namely the second and the third arguments of `qsort`, are constrained to have complementary submodes.

## 3.4 Linearity Analysis

### 3.4.1 Motivation and Observation

The mode system guarantees the uniqueness of *write* capability of each variable in a runtime configuration. Furthermore, although it does not impose any constraint on the number of read capabilities (occurrences), it imposes less generic, stronger mode constraints on programs that may discard or duplicate data. The modes of the paths where singleton variables (one-to-zero communication channels) may occur are constrained to *IN* or *OUT*, and paths of nonlinear variables (one-to-many communication channels) may very well be constrained to *IN* or *OUT*. Thus the mode system effec-

tively prefers programs that do not discard or duplicate data by giving them weaker mode constraints, providing the view of "data as resources" to some extent.

Our experiences with Prolog and concurrent logic programming show that surprisingly many variables in Prolog and concurrent logic programs are linear. For instance, all the variables in the `merge` program (Fig. 3.5) are linear, and all but one of the variables in `qsort` (Fig. 3.7) are linear. This indicates that the majority of communication is one-to-one and deserves special attention.

As a non-toy example, we examined the mode constraint solver of *klint v2*, which comprised 190 KL1 clauses [43]. Those clauses imposed 1392 constraints by Rule (BV), one for each variable in the program, of which more than 90% were of the form $m/p_1 = m/p_2$ (1074) or $m/p_1 = \overline{m/p_2}$ (183). Thus we can say that the clauses are highly linear. Furthermore, all of the 42 non-binary constraints were reduced to unary or binary constraints using other unary or binary constraints. Actually they were reduced to 6 constraints of the form $m/p_1 = \overline{m/p_2}$ and 72 constraints of the form $m/p = IN$. This means that nonlinear variables were all used under simple, one-way communication protocols.

### 3.4.2 The Linearity System

The purpose of linearity analysis [46] is to statically analyze exactly *where* nonlinear variables and shared data structures may occur – in which predicates, in which arguments, and in which part of the data structures carried by those arguments. This complements mode analysis in the sense that it is concerned with the number of *read* capabilities.

To distinguish between non-shared and shared data structures in a reduction semantics without the notion of pointers, we consider giving a linearity annotation 1 or $\omega$ to every occurrence of a constructor $f$ appearing in (initial or reduced) goal clauses and body goals in program clauses.[5] The annotations appear as $f^1$ or $f^\omega$ in the theoretical framework, though the purpose of linearity analysis is to reason about the annotations and compile them away so that the program can be executed without having to maintain linearity annotations at run time.

Intuitively, the principal constructor of a structure possibly referenced by more than one pointer must have the annotation $\omega$, while a structure always pointed to by only one pointer in its lifetime can have the annotation 1. Another view of the annotation is that it models a one-bit reference counter that is not decremented once it reaches $\omega$.

The annotations must observe the following closure condition: If the principal constructor of a term has the annotation $\omega$, all constructors occurring in the term must have the annotation $\omega$. In contrast, a term with the principal constructor annotated as 1 can contain a constructor with either annotation, which means that a subterm of a non-shared term may possibly be shared.

[5]The notation is after related work [19, 35] on different computational models.

Given linearity annotations, the operational semantics is extended to handle them so that they may remain consistent with the above intuitive meaning.

1. The annotations of constructors in program clauses and *initial* goal clauses are given according to how the structures they represent are implemented. For instance, consider the following goal clause:

   ```
   :- p([1,2,3,4,5],X), q([1,2,3,4,5],X).
   ```

   If the implementation chooses to create a single instance of the list `[1,2,3,4,5]` and let the two goals share them, the constructors (there are 11 of them including `[]`) must be given $\omega$. If two instances of the list are created and given to `p` and `q`, either annotation is compatible with the implementation.

2. Suppose a substitution $\theta = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ is applied upon one-step reduction from $Q$ to $Q'$.

   (a) When $v_i$ is nonlinear, the substitution instantiates more than one occurrence of $v_i$ to $t_i$ and makes $t_i$ shared. Accordingly, all subterms of $t_i$ become shared as well. So, prior to rewriting the occurrences of $v_i$ by $t_i$, we change all the annotations of the constructors constituting $t_i$ to $\omega$.

   (b) When $v_i$ is linear, $\theta$ does not increase the number of references to $t_i$. So we rewrite $v_i$ by $t_i$ without changing the annotations in $t_i$.

As in mode analysis, the linearity of a (well-moded) program can be characterized using a linearity function, a mapping from $P_{Atom}$ to the binary codomain $\{nonshared, shared\}$, which satisfies the closure condition

$$\forall p \in P_{Atom} \forall q \in P_{Term} (\lambda(p) = shared \Rightarrow \lambda(pq) = shared).$$

The purpose of linearity analysis is to reconstruct a linearity function (say $\lambda$) that satisfies all *linearity constraints* imposed by each program clause and a goal clause $:- B$, which are shown in Fig. 3.8. The *klint v2* system reconstructs linearity as well as mode information.

As expected, the linearity system enjoys the subject reduction theorem:

- Suppose $\lambda$ satisfies the linearity constraints of a program $P$ and a goal clause $Q$, and $Q$ is reduced in one step to $Q'$ under the extended occur-check condition. Then $\lambda$ satisfies the linearity constraints of $Q'$ as well.

An immediate consequence of the subject reduction theorem is that a constructor with $\omega$ cannot appear at $p$ such that $\lambda(p) = nonshared$. The (sole) reader of the data structure at a *nonshared* path can safely discard the top-level structure after accessing its elements. One feature of our linearity system is that it can appropriately handle

($BF_\lambda$) If a function symbol $f^\omega$ occurs at the path $p$ in $B$, then $\lambda(p) = \mathit{shared}$.

($LV_\lambda$) If a linear variable occurs both at $p_1$ and $p_2$, then

$\forall q \in P_{Term}(m(p_1 q) = \mathit{in} \wedge \lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared})$ (if $p_1$ is a head path);

$\forall q \in P_{Term}(m(p_1 q) = \mathit{out} \wedge \lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared})$ (if $p_1$ is a body path).

($NV_\lambda$) If a nonlinear variable occurs at $p$, then

$\forall q \in P_{Term}(m(pq) = \mathit{out} \Rightarrow \lambda(pq) = \mathit{shared})$ (if $p$ is a head path);

$\forall q \in P_{Term}(m(pq) = \mathit{in} \Rightarrow \lambda(pq) = \mathit{shared})$ (if $p$ is a body path).

($BU_\lambda$) For a unification body goal $=_k$, $\forall q \in P_{Term}(\lambda(\langle =_k, 1\rangle q) = \lambda(\langle =_k, 2\rangle q))$.

Fig. 3.8: Linearity constraints imposed by a clause $h$ :- | $B$

data structures whose sharing properties change (from *nonshared* to *shared*) in their lifetime, allowing update-in-place of not yet shared data structures.

There is one subtle point in this optimization. The optimization is completely safe for built-in data types such as numeric or character arrays that allow only instantiated data to be stored. However, when a structure is implemented so that its field may itself represent an uninstantiated logical variable, the structure cannot be recycled until the variable is instantiated (through an *internal pointer* to the variable) and read. Most implementations of Prolog and concurrent logic languages (including KLIC [7]) represent structures this way for efficiency reasons, in which case local reuse requires strictness analysis, namely the analysis of instantiation states of variables, in addition to linearity analysis. The implementation of KL1 on the Parallel Inference Machine disallowed internal pointers to feature local reuse based on the 1-bit reference counting technique [5].

## 3.5 From Linearity to Strict Linearity

### 3.5.1 Polarizing Constructors

We already saw that most if not all variables in concurrent logic languages are linear variables. To start another observation, consider the following insertion sort program:

```
sort([],     S) :- | S=[].
sort([X|L0],S) :- | sort(L0,S0), insert(X,S0,S).
insert(X,[],     R) :-       | R=[X].
insert(X,[Y|L], R) :- X=<Y | R=[X,Y|L].
insert(X,[Y|L0],R) :- X>Y   | R=[Y|L], insert(X,L0,L).
```

Here again, all the variables are linear (we do not count the occurrences in guard goals when considering linearity). However, an even more striking fact is that, by slight modification, all constructors (including constants) can be made to occur exactly twice as well:

```
sort([],     S) :- | S=[].
sort([X|L0],S) :- | sort(L0,S0), insert([X|S0],S).
insert([X],      R) :-       | R=[X].
insert([X,Y|L], R) :- X=<Y | R=[X,Y|L].
insert([X,Y|L0],R) :- X>Y   | R=[Y|L], insert([X|L0],L).
```

This suggests that the notion of linearity could be extended to cover constructors as well. We call it *strict linearity.* A linear variable is a *dipole* with two occurrences with opposite polarities. Likewise, a linear constructor is a dipole with two occurrences with opposite polarities, one in the head and the other in the body of a clause. The two occurrences of a linear constructor can be regarded as two *polarized* instances of the same constructor.

If all the constructors are linear in program clauses as in the second version of `sort`, all deallocated cells can be reused locally to allocate new cells without accessing a non-local free list. That is, as long as the input list is not shared, the program can construct the final result by reorganizing input cells and without generating any garbage cells or allocating new cells from non-local storage.

### 3.5.2 Strict Linearity

We say that a program clause is *strictly linear* if all the variables have exactly two channel occurrences in the clause and all the constructors have exactly two occurrences, one in the head and the other in the body.

The above definition does not require that predicate symbols occur exactly twice. If this is enforced, all body goals can inherit its goal record (that records the argument of goals) from the parent and the program can run with a fixed space, but we must have a means to terminate tail recursion, as will be discussed in Sect. 3.5.3. Although strictly linear programs still require allocation and deallocation of goal records, goal records are inherently non-shared and much more manageable than heap-allocated data. So strict linearity is a significant step toward resource-conscious programming.

Let me give another example.

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

The base case receives an empty list but does not use it. A short value such as `[]` could be regarded as a zero-resource value, but we prefer to consider $n$-ary constructors to convey $n+1$ units in general, in which case the program recovers strict linearity using an extra argument:

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

Note that the first version of append can be thought of as a *slice* of the second version.

### 3.5.3 Void: The Zero-Capability Symbol

All the examples above are transformational processes. It is much less clear how one can program reactive, server-type processes that respond to streams of requests in a strictly linear setting. Consider the following stack server:

```
stack([],         D    ) :- | true.
stack([push(X)|S],D    ) :- | stack(S,[X|D]).
stack([pop(X)|S], [Y|D]) :- | X=Y, stack(S,D).
```

One way to recover the strict linearity of this program is:

```
stack([](Z),            D) :- | Z=[](D).
stack([push([X|*],Y)|S],D) :- | Y=[push(*,*)|*], stack(S,[X|D]).
stack([pop(X,Z)|S], [Y|D]) :- | X=[Y|*], Z=[pop(*,*)|*], stack(S,D).
```

Note that an empty list, which can be regarded as an "end-of-transaction" message to a process, has been changed to a unary constructor conveying a reply box, through which the current "deposit" will be returned. Upon receiving a message, the server immediately returns the resource used by the client to send the message. In a physical mail metaphor, cons cells can be compared to envelopes and push and pop messages can be compared to cover letters, which real-world servers often fail to find a nice way of recycling.

Observe that we need to extend the language with a special symbol, *, to indicate void positions of structures. A void position will be given *zero capability* so that no read or write to the position will take place.

What is the resource aspect of variable occurrences and the void symbol? We assume that each variable occurrence uses one unit and one void symbol uses one unit. Furthermore, we assume that a non-variable term is always pointed to from a variable occurrence and an argument of a non-variable term or a goal always points to a variable occurrence. This canonical representation is not always space-optimal, but makes resource counting simple and uniform.

It is not difficult to see that the strictly linear versions of append and stack do not allocate or deallocate resources for variable occurrences and voids during tail-recursive iteration.

An interesting aspect of the *void* construct is that it could be used to recover the linearity of *predicate* symbols by allowing multi-head clauses as in Constraint Handling Rules [11]. For instance, sort in Sect. 3.5.1 could be rewritten as

```
sort([],     S) :-                  | S=[], sort(*.*).
sort([X|L0],S), insert(*,*) :- | sort(L0,S0), insert([X|S0],S).
```

where the goals with void arguments could be considered as free, inactive goals waiting for habitants. The first clause makes the current goal inactive, while the second clause explicitly says that it requires one free `insert` goal for the reduction of `sort([X|L0],S)`. However, in this paper we assume that these free goals are implicit.

Some readers will enjoy the symmetry of strictly linear programs, while others may find it cumbersome and want compilers to reconstruct strict linear versions of their programs automatically. In any case, strictly linear programs are completely recyclic and are a step towards small-footprint symbolic computation with highly predictable behavior.

One natural question is how to write programs whose output size is essentially larger than the input size. An obvious solution is to require the initial process to pass a necessary number of cells obtained from the runtime library. This will work well as long as the output size is predictable. Some programs may have the output size that is essentially the same as the input size but may require more resource to represent intermediate results. We have two solutions to this. One is to let the initial process provide all necessary resource. The other is to require that the input size and the output size be balanced for each process spawned during computation, but allow a subprocess to use (and then return) more resource than that received from the parent process. The notion of strict linearity has to be relaxed somewhat to enable the latter alternative.

### 3.5.4 Constant-Time Property of Strictly Linear Programs

The cost of the primitive operations of strictly linear concurrent logic programs are highly predictable despite their non-strict data structures.

The primitive operations of concurrent logic programs are:

1. spawning of new non-unification goals (including tail-recursive ones),
2. termination of a non-unification goal (upon reduction with a base-case clause),
3. *ask* or term matching, which may involve
   (a) synchronization, namely suspension and resumption of the process, and
   (b) pointer dereferencing, and
4. *tell*, namely execution of a unification body goal, which may involve pointer dereferencing.

On a single-processor environment, spawning and termination of a goal involves (de)allocation of a fixed-size goal record and manipulation of a goal queue, which can both be regarded as constant-time operations.

Synchronization involves the hooking and unhooking of goals on an uninstantiated variable, but in a linear setting, the number of goals hooked on each variable is at most one.

The cost of dereferencing reflects the length of the chain of pointers formed by unification. Due to the flexibility logical variables bring in the way of data structure formation, even in sequential Prolog it is possible to create an arbitrarily long chain of pointers between variables.

In a linear setting, however, every uninstantiated variable has exactly two occurrences. We can represent it using two cells that form a size-two cycle by pointing to each other. Then the unification of two linear variables, say $v_1$ and $v_2$, which consumes one $v_1$ and one $v_2$ by the unification goal itself, can be implemented by letting the other occurrence of $v_1$ and the other occurrence of $v_2$ point to each other. This keeps the size-two property of uninstantiated linear variables unchanged. The writing to a linear variable $v$, which consumes one occurrence of $v$, dereferences the pointer to reach the other occurrence of $v$ and instantiate it. The reader of $v$ dereferences it exactly once to access its value.

## 3.6 Allowing Concurrent Access within Strict Linearity

Strict linearity can be checked by slightly extending the mode and linearity systems described earlier. However, rather than doing so, we consider extending the framework to allow concurrent access to shared resource. There are two reasonable ways of manipulating resource such as large arrays concurrently.

One is to give different processes exclusive (i.e., non-shared) read/write capabilities to *different* parts of a data structure. This is easily achieved by (i) splitting a non-shared structure, (ii) letting each process work on its own fragment and return results by update-in-place, and (iii) joining the results into one. For instance, parallel quicksort of an array has been implemented this way using optimized versions of KLIC's vectors [25]. Concurrent manipulation of this type fits nicely within the present framework of mode and linearity systems because no part of an array becomes shared.

On the other hand, some applications require concurrent accesses with non-exclusive, read capability to the whole of a data structure to allow concurrent table lookup and so on. When the accesses can be sequentialized, the structure with an exclusive capability can be returned finally either

1. by letting each process receive and return an exclusive capability one after another or

2. by guaranteeing the sequentiality of accesses by other language constructs (`let`, guard, *etc.*) as in [49] and [18].

However, these solutions cannot be used when we need to run the readers concurrently or in parallel. Our goal is to allow some process (say $P$) to collect all released non-

exclusive capabilities so that $P$ may restore an *exclusive* capability and update it in place.

For this purpose, we refine the $\{in, out\}$ capability domain of the mode system to a continuous domain $[-1, +1]$. As in the mode system, the capability is attached to all paths. Let $\kappa$ be the capability of the principal constructor of some occurrence of a variable in a configuration. Then

1. $\kappa = -1$ means 'exclusive write',
2. $-1 < \kappa < 0$ means 'non-exclusive write',
3. $\kappa = 0$ means no capability,
4. $0 < \kappa < 1$ means 'non-exclusive read', and
5. $\kappa = +1$ means 'exclusive read'.

This is a refinement of the mode system in the sense that *out* corresponds to $-1$ and *in* corresponds to $(0, +1]$. This is also a refinement of the linearity system in the sense that *nonshared* corresponds to $\pm 1$ and *shared* corresponds to $(0, +1]$.

Then, what meaning should we give to the $(-1, 0]$ cases? Suppose a `read` process receives an exclusive read capability to access `X0` and split the capability to two non-exclusive capabilities using the following clause:

```
read(X0,X) :- | read1(X0,X1), read2(X0,X2), join(X1,X2,X).
```

The capability `X0` conveys can be written as a function $\mathbf{1}$, a constant function such that $\mathbf{1}(p) = +1$ for all $p \in P_{Atom}$. We don't care how much of the $\mathbf{1}$ capability goes to `read1` but just require that the capabilities of the two `X0`'s sum up to $\mathbf{1}$. Different paths in one of the split occurrences of `X0` may convey different capabilities. Also, we assume that the capabilities given to `read1` and `read2` are returned with the opposite polarity through `X1` and `X2`. Logically, `X1` and `X2` will become the same as `X0`. Then, `join` process defined as

```
join(A,A,B) :- | B = A.
```

checks if the first two arguments are indeed aliases and then returns it through the third argument. Note the interesting use of a nonlinear head. The capability constraint for the join program is that the capabilities of the three arguments must sum up to a constant function $\mathbf{0}$.

Now the `X` returned by `join` is guaranteed to convey the $\overline{\mathbf{1}}$ (exclusive write) capability that complements the capability received through `X0`.

## 3.7 Operational Semantics with Capability Counting

In the linearity analysis described earlier, the operational semantics was augmented with a linearity annotation 1 or $\omega$ given to every occurrence of a constructor $f$ appearing in (initial or reduced) goal clauses. Here, we replace the annotation with a capability annotation $\kappa$ ($0 < \kappa \leq 1$). $\kappa = 1$ (exclusive) corresponds to the 1 annotation meaning 'non-shared', while $\kappa < 1$ (non-exclusive) refines (the reciprocal of) $\omega$. Again, the annotations are to reason about capabilities statically and are to be compiled away.

The annotations must observe the following closure condition: If the principal constructor of a term has a non-exclusive capability, all constructors occurring in the term must have non-exclusive capabilities as well. In contrast, a term with an exclusive principal constructor can contain a constructor with any capability.

The operational semantics is extended to handle annotations so that they may remain consistent with the above intuitive meaning. However, before doing so, let us consider how we can start computation within a strictly linear framework. The goal clause

```
:- sort([3,1,4,1,5,9],X).
```

is not an ideal form to work with because variables and constructors are monopoles. Instead, we consider a strictly linear version of the goal clause

```
main([3,1,4,1,5,9],X) :- | sort([3,1,4,1,5,9],X).
```

in which the head complements the resources in the body. The head declares the resources necessary to initiate computation and the resources to be returned to the environment. The reduction semantics works on the body goal as before, except that the unification goal to instantiate `X` remains intact. Then the above clause will be reduced to

```
main([3,1,4,1,5,9],X) :- | X = [1,1,3,4,5,9].
```

which can be thought of as a normal form in our polarized setting.

Hereafter, we assume that an initial goal clause is complemented by a head to make it strictly linear.

1. All the constructors in the body of an *initial* goal clause are given the annotation 1. This could be relaxed as we did in giving the linearity annotations (Sect. 3.4) to represent initially shared data, but without loss of generality we can assume that initial data are non-shared.

2. Suppose a substitution $\theta = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ is applied upon one-step reduction from $Q$ to $Q'$.

   (a) When $v_i$ is nonlinear,[6] the substitution instantiates more than one occurrence of $v_i$ to $t_i$ and makes $t_i$ shared. Accordingly, all the subterms of $t_i$

[6]The term '*sublinear*' might be more appropriate than *nonlinear* here.

become shared as well. So, prior to rewriting the occurrences of $v_i$ by $t_i$, we change all the annotations of the constructors constituting $t_i$ as follows: Let $f^\kappa$ be a constructor in $t_i$ and $t_i$ is about to be copied to $m$ places (this happens when $v_i$ occurs $m+1$ times in the goal clause). Then $\kappa$ is split into $\kappa_1, \ldots, \kappa_m$, where $\kappa_1 + \ldots + \kappa_m = \kappa$, $\kappa_j > 0$ $(1 \leq j \leq m)$, and the $\kappa_j$'s are mutually different and not used previously as capabilities.

(b) When $v_i$ is linear, $\theta$ does not increase the number of references to $t_i$. So we rewrite $v_i$ by $t_i$ without changing the annotations in $t_i$.

Furthermore, to deal with capability polymorphism described later, we index the predicate symbols of the goals in an initial goal clause with 1, 2, and so on. The indices are in fact singleton sequences of natural numbers which will be extended in each reduction. That is, when reducing a non-unification goal $b_s$ ($s$ being the index) to spawn $b^1, \ldots, b^n$ using Rule (iii) of Fig. 3.2, the new goals are indexed as $b^1_{s.1}, \ldots, b^n_{s.n}$.

## 3.8 The Capability System

Our capability system generalizes the mode system. As suggested earlier, the capability type (say $c$) of a program or its fragment is a function

$$c : P_{Atom} \rightarrow [-1, +1].$$

The framework is necessarily polymorphic with respect to non-exclusive capabilities because a non-exclusive capability may be split into two or more capabilities. This is why different goals created at runtime should be distinguished using indices.

The following closure conditions of a capability function represent the uniformity of non-exclusive capabilities:

1. $0 < c(p) < 1 \Rightarrow \forall q(0 < c(pq) < 1)$
2. $-1 < c(p) < 0 \Rightarrow \forall q(-1 < c(pq) < 0)$

Our capability constraints, shown in Fig. 3.9, generalizes mode constraints (Fig. 3.4) without complicating it. Here we have inherited all the notational conventions from the mode system (see Sect. 3.3.3) and modified them appropriately.

As an example, consider the following program.

```
p_s(X,Y,...) :- | r_s.1(X,Y1,...), p_s.2(X,Y2,...), join_s.3(Y1,Y2,Y).
p_s(X,Y,...) :- | X=_s.1 Y.
join_s(A,A,B) :- | B=_s.1 A.
```

Then the capability constraints they impose include:

($\mathrm{BU}_c$) $\forall s \forall t_1, t_2 \in \mathit{Term}((t_1 =_s t_2) \in B \Rightarrow c/\langle =_s, 1\rangle + c/\langle =_s, 2\rangle = \mathbf{0})$
(the arguments of a unification body goal have complementary capabilities)

($\mathrm{BV}_c$) Let $v \in \mathit{Var}$ occur $n\ (\geq 1)$ times in $h$ and $B$ at $p_1, \ldots, p_n$, of which the occurrences in $h$ are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

1. $-c/p_1 - \ldots - c/p_k + c/p_{k+1} + \ldots + c/p_n = \mathbf{0}$ *(Kirchhoff's Current Law)*
2. if $k = 0$ and $n > 2$ then $\mathcal{R}(\{c/p_1, \ldots, c/p_n\})$
3. if $k \geq 1$ and $n - k \geq 2$ then $\mathcal{R}(\{\overline{c/p_1}, c/p_{k+1}, \ldots, c/p_n\})$

where $\mathcal{R}$ is a 'cooperativeness' relation:

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \exists s \in S(s < \mathbf{0} \wedge \forall s' \in S \backslash \{s\}\,(s' > \mathbf{0}))$$

($\mathrm{HV}_c$) $\forall p \in P_{\mathit{Atom}}(\widetilde{h}(p) \in \mathit{Var} \wedge \exists p' \neq p(\widetilde{h}(p) = \widetilde{h}(p')) \Rightarrow c/p > \mathbf{0})$
(if the symbol at $p$ in $h$ is a variable occurring elsewhere in $h$, then $c/p > \mathbf{0}$)

($\mathrm{HF}_c$) $\forall p \in P_{\mathit{Atom}}(\widetilde{h}(p) \in \mathit{Fun} \Rightarrow (c(p) > 0$
$\wedge \exists! q \in P_{\mathit{Atom}} \exists a \in B(\widetilde{a}(q) = \widetilde{h}(p) \wedge c(p) = c(q) \wedge (c(p) < 1 \Rightarrow c/p = c/q))))$
(if the symbol at $p$ in $h$ is a constructor, $c(p) > 0$ and there's exactly one partner in $B$ at $q$ such that $c(p) = c(q)$ (and $c/p = c/q$ if non-exclusive))

($\mathrm{BF}_c$) $\forall p \in P_{\mathit{Atom}} \forall a \in B(\widetilde{a}(p) \in \mathit{Fun} \Rightarrow (c(p) > 0$
$\wedge \exists! q \in P_{\mathit{Atom}}(\widetilde{h}(q) = \widetilde{a}(p) \wedge c(p) = c(q) \wedge (c(p) < 1 \Rightarrow c/p = c/q))))$
(if the symbol at $p$ in $B$ is a constructor, $c(p) > 0$ and there's exactly one partner at $q$ in $h$ such that $c(p) = c(q)$ (and $c/p = c/q$ if non-exclusive))

($\mathrm{Z}_c$) $\forall p \in P_{\mathit{Atom}} \forall a \in B((\widetilde{h}(p) = * \vee \widetilde{a}(p) = *) \Rightarrow c/p = \mathbf{0})$
(a void path has a zero capability)

($\mathrm{NZ}_c$) $\forall p \in P_{\mathit{Atom}} \forall a \in B((\widetilde{h}(p) \in \mathit{Fun} \cup \mathit{Var} \vee \widetilde{a}(p) \in \mathit{Fun} \cup \mathit{Var}) \Rightarrow c(p) \neq 0)$
(a non-void path has a non-zero capability)

Fig. 3.9: Capability constraints imposed by a clause $h$ :- | $B$

1. From the first clause of p:
   (a) $-c/\langle \mathtt{p}_s, 1\rangle + c/\langle \mathtt{r}_{s.1}, 1\rangle + c/\langle \mathtt{p}_{s.2}, 1\rangle = \mathbf{0}$ (by ($\mathrm{BV}_c$) applied to X)
   (b) $c/\langle \mathtt{r}_{s.1}, 2\rangle + c/\langle \mathtt{join}_{s.3}, 1\rangle = \mathbf{0}$ (by ($\mathrm{BV}_c$) applied to Y1)
   (c) $c/\langle \mathtt{p}_{s.2}, 2\rangle + c/\langle \mathtt{join}_{s.3}, 2\rangle = \mathbf{0}$ (by ($\mathrm{BV}_c$) applied to Y2)
   (d) $-c/\langle \mathtt{p}_s, 2\rangle + c/\langle \mathtt{join}_{s.3}, 3\rangle = \mathbf{0}$ (by ($\mathrm{BV}_c$) applied to Y)

2. From the second clause of p:
   (a) $-c/\langle \mathtt{p}_s, 1\rangle + c/\langle =_s, 1\rangle = \mathbf{0}$ (by ($\mathrm{BV}_c$) applied to X)
   (b) $-c/\langle \mathtt{p}_s, 2\rangle + c/\langle =_s, 2\rangle = \mathbf{0}$ (by ($\mathrm{BV}_c$) applied to Y)
   (c) $c/\langle =_s, 1\rangle + c/\langle =_s, 2\rangle = \mathbf{0}$ (by ($\mathrm{BU}_c$))

3. From join:
   (a) $c/\langle \mathtt{join}_s, 1\rangle > \mathbf{0}$ (by ($\mathrm{HV}_c$) applied to A)
   (b) $c/\langle \mathtt{join}_s, 2\rangle > \mathbf{0}$ (by ($\mathrm{HV}_c$) applied to A)
   (c) $-c/\langle \mathtt{join}_s, 1\rangle - c/\langle \mathtt{join}_s, 2\rangle + c/\langle =_s, 2\rangle = \mathbf{0}$ (by ($\mathrm{BV}_c$) applied to A)
   (d) $-c/\langle \mathtt{join}_s, 3\rangle + c/\langle =_s, 1\rangle = \mathbf{0}$ (by ($\mathrm{BV}_c$) applied to B)
   (e) $c/\langle =_s, 1\rangle + c/\langle =_s, 2\rangle = \mathbf{0}$ (by ($\mathrm{BU}_c$))

In each constraint, the index $s$ is universally quantified. These constraints are satisfiable if (and only if) $c/\langle \mathtt{r}_{s.1}, 1\rangle + c/\langle \mathtt{r}_{s.1}, 2\rangle = \mathbf{0}$. Suppose this can be derived from other constraints. Suppose also that $c/\langle p_{s_0}, 1\rangle = \mathbf{1}$ holds, that is, p is initially called with a non-shared, read-only first argument. Then the above set of constraints guarantees $c/\langle p_{s_0}, 2\rangle = \overline{\mathbf{1}}$, which means that the references to X distributed to the r's will be fully collected as long as all the r's eventually return the references they have received.

Note that the above constraints (1(a) and several others) and Rule ($\mathrm{NZ}_c$) entail $\mathbf{0} < c/\langle \mathtt{r}_{s.1}, 1\rangle < \mathbf{1}$ and $\mathbf{0} < c/\langle \mathtt{p}_{s.2}, 1\rangle < \mathbf{1}$. That is, these paths are constrained to be non-exclusive paths. It is easy to see that a set of constraints cannot entail a constraint of the form $\mathbf{0} < c/p < \mathbf{1}$ unless some variable is nonlinear.

We have not yet worked out on theoretical results, but conjecture that the following properties hold (possibly with minor modification):

1. The three properties shown in Sect. 3.3.5, namely (i) degeneration of unification to assignment, (ii) subject reduction, and (iii) groundness.

2. (*Conservation of Constructors*) A reduction does not gain or lose any constructor in the goal clause, with its capability taken into account as its weight.

The Rules ($\mathrm{HF}_c$) and ($\mathrm{BF}_c$) can be relaxed so that the name of the constructor examined in the head can be changed when it is recycled in the body, as long as the constructor comes with an exclusive capability and its arity does not change. When this modification is done, the Conservation of Constructors property should be modified accordingly to allow the changes of names.

This modification is important when computation involves a lot of constants such as numbers. Indeed, some relaxation will be necessary to accommodate arithmetics in our framework in a reasonable way. For instance, to perform local computation such as

`Y:=X+2`, it would be unrealistic to obtain constructors + and 2 from the parent process and let them escape through `Y`. Rather, we want to allocate and garbage-collect them locally and let `Y` emit an integer constant.[7]

## 3.9 Related Work

Relating the family of $\pi$-calculi and the CCP formalism has been done as proposals of calculi such as the $\gamma$-calculus [33], the $\rho$-calculus [24] and the Fusion calculus [48], all of which incorporate constraints (or name equation) in some form. The $\gamma$-calculus is unique in that it uses procedures with encapsulated states to model concurrency and communication rather than the other way around. The $\rho$-calculus introduces constraints into name-based concurrency, while constraint-based concurrency aims to demonstrate that constraints alone are adequate for modeling and programming concurrency. The Fusion calculus simplifies the binding operators of the $\pi$-calculus using the unification of names. A lesson learned from Constraint Logic Programming [17] is that, even when general constraint satisfaction is not intended, formulation in terms of constraints can be more elegant and less error-prone. The simplicity of constraint-based concurrency and the existence of working implementations suggest that encoding all these calculi in constraint-based concurrency would be worthwhile.

In addition to $\rho$ and Fusion, various calculi based on the $\pi$-calculus have been proposed, which include L$\pi$ (Local $\pi$) [22], the Join calculus [10] and $\pi$I (Internal $\pi$) [26]. They are aimed at nicer semantical properties and/or better correspondence to programming constructs. Some of the motivations of these calculi are in common – at least at a conceptual level – with the design of constraint-based concurrency with strong moding. For instance, $\pi$I restricts communicated data to local names in order to control name scope, and L$\pi$ restricts communicated data to those with output capabilities in order to allow names to act as object identities. Both objectives have been achieved in constraint-based concurrency. L$\pi$ abolished name matching based on the observation that it would be too strong a capability. The counterpart of name matching in constraint-based concurrency is matching with a nonlinear head, which imposes a strong mode constraint that bans the comparison of channels used for bidirectional communication.

In concurrent, logic, and/or functional languages and calculi, a number of type systems to deal with polarities and linearities have been proposed.

In $\pi$-calculi and functional languages, Kobayashi proposes a linear type system for the $\pi$-calculus [19], which seems to make the calculus close to constraint-based concurrency with linear, moded variables because both linear channels and linear logic variables disallow more than one write access and more than one read access. Turner *et al.* introduce linearity annotation to a type system for call-by-need lambda calculus

[7] In actual implementations, + and 2 will be embedded in compiled code and can be considered zero-resource values.

[35]. All these pieces of work could be considered the application of ideas with similar motivations to different computational models. In concurrent logic programming, the difficulty lies in the treatment of arbitrarily complex information flow expressed using logical variables. Walker discusses types supporting more explicit memory management [50]. Session types [13] shares the same objective with our mode system.

Languages that feature linearity can be found in various programming paradigms. Linear Lisp [4] and Lilac [20] are two examples outside logic programming, while a survey of linear logic programming languages can be found in [23].

There is a lot of work on compile-time garbage collection other than that based on typing. In logic programming, most of the previous work is based on abstract interpretation [14]. Mercury [34] is a logic programming language known for its high-performance and enables compile-time garbage collection using mode and uniqueness declarations [21]. However, the key difference between Mercury and GHC is that the former does not allow non-strict data structures while the latter is highly non-strict.

Message-oriented implementation of Moded Flat GHC, which compiles stream communication into tight control flow between communicating processes, can be thought of as a form of compile-time garbage collection [41][40]. Another technique related to compile-time garbage collection is process fusion by unfold/fold transformation [38], which should have some relationship with deforestation of functional programs.

Janus [27] establishes the linearity property by allowing each variable to occur only twice. In Janus, a reserved unary constructor is used to give a variable occurrence an output capability. Our technique allows both linear and nonlinear variables and distinguishes between them by static analysis, and allows output capabilities to be inferred rather than specified.

Concurrent read accesses under linear typing was motivated by the study on parallel array processing in Moded Flat GHC [42] [25], which again has an independent counterpart in functional programming [29].

## 3.10 Conclusions and Future Work

This is the first report on the ongoing project on garbage-free symbolic computation based on constraint-based concurrency.

The sublanguage we propose, namely a strictly linear subset of Guarded Horn Clauses, retains most of the power of the cooperative use of logical variables, and also allows resource sharing without giving up the linguistic-level control over the resource handled by the program.

The capability type system integrates and generalizes the mode system and the linearity system developed and used for Flat GHC. Thanks to its arithmetic and constraint-based formulation, the type system is kept quite simple. We plan to build a constraint-based type reconstructor in the near future. A challenging issue from the theoretical point of view is the static analysis of the extended occur-check condition. However, we

have already been successful in detecting the (useless) unification of identical nonlinear variable as erroneous; if `X` is unified with itself when it has the third occurrence elsewhere, the third occurrence is constrained to have zero capability, which contradicts Rule ($NZ_c$). Another important direction related to resource-consciousness is to deal with time as well as space bounds. We need to see how type systems developed in different settings to deal with resource bounds [16][9] can relate to our concurrent setting.

Undoubtedly, the primary concern is the ease of programming. Does resource-conscious programming help programmers write correct programs enjoying better properties, or is it simply burdensome? We believe the answer to the former is at least partly affirmative, but to a varying degree depending on the applications. One of the grand challenges of concurrent languages and their underlying theories is to provide a common platform for various forms of non-conventional computing including parallel computing, distributed/network computing, real-time computing, and mobile computing [45]. All these areas are strongly concerned with physical aspects and we hope that a flexible framework with the notion of resources will be a promising starting point towards a common platform.

# References


[1] Aït-Kaci, H. and Nasr, R., LOGIN: A Logic Programming Language with Built-In Inheritance. *J. Logic Programming*, Vol. 3, No. 3 (1986), pp. 185–215.

[2] Ajiro, Y., Ueda, K. and Cho, K., Error-Correcting Source Code. In *Proc. Fourth Int. Conf. on Principles and Practice of Constraint Programming (CP'98)*, LNCS 1520, Springer-Verlag, 1998, pp. 40–54.

[3] Ajiro, Y. and Ueda, K., Kima: an Automated Error Correction System for Concurrent Logic Programs. To appear in *Automated Software Engineering*, 2001.

[4] Baker, H. G., Lively Linear Lisp—'Look Ma, No Garbage!' *Sigplan Notices*, Vol. 27, No. 8 (1992), pp. 89–98.

[5] Chikayama, T. and Kimura, Y., Multiple Reference Management in Flat GHC. In *Logic Programming: Proc. of the Fourth Int. Conf. (ICLP'87)*, The MIT Press, 1987, pp. 276–293.

[6] Chikayama, T., Operating System PIMOS and Kernel Language KL1. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992 (FGCS'92)*, Ohmsha and IOS Press, Tokyo, 1992, pp. 73–88.

[7] Chikayama, T., Fujise, T. and Sekita, D., A Portable and Efficient Implementation of KL1. In *Proc. 6th Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'94)*, LNCS 844, Springer-Verlag, 1994, pp. 25–39.

[8] Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.

[9] Crary, K. and Weirich, S., Resource Bound Certification. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL'00)*, 2000, pp. 184–198.

[10] Fournet, C., Gonthier, G. Lévy, J.-J., Maranget, L. and Rémy, D., A Calculus of Mobile Agents. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, LNCS 1119, Springer-Verlag, 1996, pp. 406–421.

[11] Frühwirth, T., Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.

[12] Fujita, H. and Hasegawa, R., A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm. In *Proc. Eighth Int. Conf. on Logic Programming (ICLP'91)*, The MIT Press, Cambridge, MA, 1991, pp. 535–548.

[13] Gay, S. and Hole, M., Types and Subtypes for Client-Server Interactions. In *Proc. European Symp. on Programming (ESOP'99)*, LNCS 1576, Springer-Verlag, 1999, pp. 74–90.

[14] Gudjonsson, G. and Winsborough, W. H., Compile-time Memory Reuse in Logic Programming Languages Through Update in Place. *ACM Trans. Prog. Lang. Syst*, Vol. 21, No. 3 (1999), pp. 430–501.

[15] Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication. In *Proc. Fifth Conf. on Object-Oriented Programming (ECOOP'91)*, LNCS 512, Springer-Verlag, 1991, pp. 133–147.

[16] Hughes, J. and Pareto, L., Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *Proc. Fourth ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'99)*, 1999, pp. 70–81.

[17] Jaffar, J. and Maher, M. J., Constraint Logic Programming: A Survey. *J. Logic Programming*, Vol. 19–20 (1994), pp. 503–582.

[18] Kobayashi, N., Quasi-Linear Types In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL'99)*, ACM, 1999, pp. 29–42.

[19] Kobayashi, N., Pierce, B. and Turner, D., Linearity and the Pi-Calculus. *ACM Trans. Prog. Lang. Syst.*, Vol. 21, No. 5 (1999), pp. 914–947.

[20] Mackie, I., Lilac: A Functional Programming Language Based on Linear Logic. *J. Functional Programming*, Vol. 4, No. 4 (1994), pp. 1–39.

[21] Mazur, N., Janssens, G. and Bruynooghe, M., A Module Based Analysis for Memory Reuse in Mercury. In *Proc. Int. Conf. on Computational Logic (CL2000)*, LNCS 1861, Springer-Verlag, 2000, pp. 1255–1269.

[22] Merro, M., Locality in the $\pi$-calculus and Applications to Distributed Objects. PhD Thesis, Ecol des Mines de Paris, 2000.

[23] Miller, D., A Survey on Linear Logic Programming. *The Newsletter of the European Network in Computational Logic*, Vol. 2, No. 2 (1995), pp.63–67.

[24] Niehren, J. and Müller, M., Constraints for Free in Concurrent Computation. In *Proc. Asian Computing Science Conf. (ACSC'95)*, LNCS 1023, Springer-Verlag, 1995, pp. 171–186.

[25] Sakamoto, K., Matsumiya, S. and Ueda, K., Optimizing Array Processing of Parallel KLIC. In *IPSJ Trans. on Programming*, Vol. 42, No. SIG 3(PRO 10) (2001), pp. 1–13 (in Japanese).

[26] Sangiorgi, D., $\pi$-Calculus, Internal Mobility and Agent-Passing Calculi. *Theoretical Computer Science*, Vol. 167, No. 1–2 (1996), pp. 235–274.

[27] Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conf. on Logic Programming (NACLP'90)*, The MIT Press, Cambridge, MA, 1990, pp. 431–446.

[28] Saraswat, V. A. and Rinard, M., Concurrent Constraint Programming (Extended Abstract). In *Proc. 17th Annual ACM Symp. on Principles of Programming Languages (POPL'90)*, ACM, 1990, pp. 232–245.

[29] Sastry, A. V. S. and Clinger, W., Parallel Destructive Updating in Strict Functional Languages. In *Proc. 1994 ACM Conf. on LISP and Functional Programming*, 1994, pp. 263–272.

[30] Shapiro, E. Y., Concurrent Prolog: A Progress Report. *IEEE Computer*, Vol. 19, No. 8 (1986), pp. 44–58.

[31] Shapiro, E., The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.

[32] Shapiro, E. Y., Warren, D. H. D., Fuchi, K., Kowalski, R. A., Furukawa, K., Ueda, K., Kahn, K. M., Chikayama, T. and Tick, E., The Fifth Generation Project: Personal Perspectives. *Comm. ACM*, Vol. 36, No. 3 (1993), pp. 46–103.

[33] Smolka, G., A Foundation for Higher-order Concurrent Constraint Programming. In *Proc. First Int. Conf. on Constraints in Computational Logics*, LNCS 845, Springer-Verlag, 1994, pp. 50–72.

[34] Somogyi, Z., Henderson, F. and Conway, T., The Execution Algorithm of Mercury, An Efficient Purely Declarative Logic Programming Language. *J. Logic Programming*, Vol. 29, No. 1–3 (1996), pp. 17–64.

[35] Turner, D. N., Wadler, P. and Mossin, C., Once Upon a Type. In *Proc. Seventh Int. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, ACM, 1995, pp. 1–11.

[36] Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming '85*, Wada, E. (ed.), LNCS 221, Springer-Verlag, 1986, pp. 168–179.

[37] Ueda, K., Guarded Horn Clauses. D. Eng. Thesis, Univ. of Tokyo, 1986.

[38] Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988 (FGCS'88)*, ICOT, Tokyo, 1988, pp. 582–591.

[39] Ueda, K. and Chikayama, T. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.

[40] Ueda, K. and Morita, M., Message-Oriented Parallel Implementation of Moded Flat GHC. *New Generation Computing*, Vol. 11, No. 3–4 (1993), pp. 323–341.

[41] Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.

[42] Ueda, K., Moded Flat GHC for Data-Parallel Programming. In *Proc. FGCS'94 Workshop on Parallel Logic Programming*, ICOT, Tokyo, 1994, pp. 27–35.

[43] Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, LNCS 1068, Springer-Verlag, 1996, pp. 134–153.

[44] Ueda, K., *klint* — Static Analyzer for KL1 Programs. Available from `http://www.icot.or.jp/ARCHIVE/Museum/FUNDING/funding-98-E.html`, 1998.

[45] Ueda, K., Concurrent Logic/Constraint Programming: The Next 10 Years. In *The Logic Programming Paradigm: A 25-Year Perspective*, Apt, K. R., Marek, V. W., Truszczynski M., and Warren D. S. (eds.), Springer-Verlag, 1999, pp. 53–71.

[46] Ueda, K., Linearity Analysis of Concurrent Logic Programs. In *Proc. Int. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, 2000, pp. 253–270.

[47] van Emden, M. H. and de Lucena Filho, G. J., Predicate Logic as a Language for Parallel Programming. In *Logic Programming*, Clark, K. L. and Tärnlund, S. -Å. (eds.), Academic Press, London, 1982, pp. 189–198.

[48] Victor, B., The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes, PhD Thesis, Uppsala Univ., 1998.

[49] Wadler, P., Linear Types Can Change the World! In *Prof. IFIP TC2 Working Conf. on Programming Concepts and Methods*, Broy, M. and Jones, C. (eds.), North-Holland, 1990, pp. 347–359.

[50] Walker, D. P., Typed Memory Management. PhD thesis, Cornell Univ., 2001.

# 第4章　並行論理プログラムの逐次性解析

We present a bottom-up method of extracting those fragments of concurrent logic programs that can be executed sequentially, and we also propose a framework of optimizing compilation of concurrent logic programs that uses sequential intermediate code generated with the method. The method is directed by the inference of certain properties on sequentiality, called *interfaces*, of concurrent processes. We formalize them in terms of the operational semantics, which enables us to formally justify the inference laws on interfaces used in the analysis.

The specialization of concurrent processes is critical for an optimizing compiler, for it can reduce the runtime overhead of process management and of data manipulation substantially. The inference laws on interfaces ensure the compositional generation of the sequential specialized code that enables such optimization. Moreover, a formal definition of the intermediate code will prove the correctness of some implementation-level optimization techniques, including tag elimination and update-in-place optimization, that cannot be formally proved by source-level analysis.

Although the formalism of interfaces exploits the constraint-based communication feature of the language, our method can be applied, in principle, to the extraction of sequentiality in other fine-grained concurrent languages.

## 4.1 Introduction

### 4.1.1 Background

Fine-grained concurrent languages such as concurrent logic programming languages [9] allow us to describe parallel computation in a natural fashion. In general, a runtime system for these languages that should deal with nondeterminacy of the program tends to suffer from the overhead of context switching and runtime data manipulation. However, because of physical limitations some fragments of a program can be executed more efficiently by compiling them into sequential code, and compilation into sequential code enables various low-level optimization techniques that reduce runtime overhead substantially.

As an example, suppose we can infer from Figure 4.1 that `intlist` and `sum` can be reduced alternately. Then an optimizing compiler can perform elimination of suspension check (as well as suspension itself), tag elimination (or unboxing), and less heap usage for local communication.

```
stair(N,X0,X) :- true | intlist(1,N,S), sum(S,X0,X).
intlist(K0,N,S) :- K0 >=N | S=[].
intlist(K0,N,S) :- K0 < N | S=[K0|S1], K:=K0+1, intlist(K,N,S1).
sum([], X0,X) :- true | X=X0.
sum([E|S],X0,X) :- true | X1:=X0+E, sum(S,X1,X).
```

Fig. 4.1: A Concurrent Logic Program.

In principle, such optimization can be achieved by dataflow analysis that decides whether a process can be executed without suspension under a given input constraint. Previous work on such optimization includes dependence analysis between goals [7], abstract interpretation [2], and demand transformation analysis [6]. But they all suffer from nondeterminism in the program and cannot always analyze the sequentiality for the sequentializing optimization purposes. For instance, because the goal `stair(0,X0,X)` tells `X0=X`, `stair` does not depend on its second argument. Thus code not specialized for the case where it has integer must incur the suspension overhead and may well lose opportunities for tag elimination.

### 4.1.2 The Proposed Framework

Our challenge is to formalize the sequentialization process, and apply it to the construction of an optimizing compiler whose correctness is formally justified. In this paper, we propose a framework for constructing an optimizing compiler for concurrent languages that

1. extracts sequentiality by the sequentiality analysis,
2. generates sequential intermediate code that is specialized for the extracted sequential usage, and
3. performs various code optimization including tag elimination and update-in-place optimization.

Our sequentiality analysis is directed by the inference of interfaces of a process. Roughly speaking, an *interface* tells us a property of a process that under a class of input constraints what class of constraints it can tell without suspension and as what agent the residual process behaves. The basic idea is to regard sequentializing specialization of a process as the selection of an interface of the process.

We have chosen concurrent logic languages to explain our framework, for it features constraint-based communication, which enables us to formalize our analysis in a concise way. However, our framework can be applied to the optimizing compilation of other fine-grained concurrent languages, such as pi-calculus, to achieve sequentializing specialization to some extent.

The rest of this paper is organized as follows. Section 4.2 defines a concurrent logic language and its operational semantics we will work on. Section 4.3 formalizes an interface in terms of the operational semantics. Sections 4.4 and 4.5 explain our framework of optimizing compilation that generates intermediate code directed by the bottom-up analysis using interfaces. Section 4.6 mentions the optimization of the generated intermediate code. Section 4.7 shows some related work, and Section 4.8 concludes.

## 4.2 The Language

This section defines a concurrent logic language we will work on. In concurrent logic programming, computation is essentially a series of asks and tells of constraints with a shared constraint store that is strengthened monotonically. We borrow the formalization of CCP (concurrent constraint programming) [8] to represent the behavior of a process in a modular way.

### 4.2.1 The Store

Let $\langle Con, \leq \rangle$ be a lattice with the bottom element *true* and the top element *false*. We call an element of *Con* a *constraint*. We assume that the sets of *variables* and *terms* of first-order logic are given and that every constraint can be represented as a term. For $c$, $d \in Con$, let the least upper bound $c \sqcup d$ represent their conjunction. In other words, $c \leq d$ means that $d$ implies $c$. A store can be represented as a constraint. For example, the store $c$ is strengthened to $c \sqcup d$ by a tell of $d$.

For every variable $X$ and positive integer $i$, we assume there exists a variable, denoted by $X \cdot i$, that is intended to represent the $i$-th argument of the term unified to $X$.

A variable $X$ is said to *capture* the term $t$ if there exists a variable $X \cdot i_1 \cdot \ldots \cdot i_n$ that syntactically occurs in $t$ with some $n \geq 0$. We say a variable is a *root variable* if no other variable captures it. We assume that for every variable $Y$ there exists exactly one root variable that captures $Y$. We also assume that there exists an infinite number of

```
  stair :: intlist(1,α·1,S) || sum(S,α·2,α·3)
intlist :: ask(α·1 >= α·2) → tell(α·3 = [])
         + ask(α·1 < α·2)
              → ( tell(α·3 = [α·1|S]) || add(α·1,1,K) || intlist(K,α·2,S) )
    sum :: ask(func(α·1,[],0)) → tell(α·3 = α·2)
         + ask(func(α·1,.,2)) → ( add(α·2,α·1·1,X) || sum(α·1·2,X,α·3) )
```

Fig. 4.4: A Program Translated from Figure 4.1.

**Axioms.**

(X1) $\exists_X c \leq c$

(X2) $c \leq d \Rightarrow \exists_X c \leq \exists_X d$

(X3) $\exists_X \exists_Y c = \exists_Y \exists_X c$

(X4) $\exists_X (c \sqcup \exists_X d) = \exists_X c \sqcup \exists_X d$

(A1) $\exists_X c = c$ if $X$ doesn't capture $c$

(A2) $\exists_X (X = Y) = \mathit{true}$

(A3) $\exists_X \mathtt{func}(X, f, n) = \mathit{true}$

(U1) $(X = X) \leq \mathit{true}$

(U2) $(X = Y) \leq (Y = X)$

(U3) $c[Y/X] \leq c \sqcup (X = Y)$

(P1) $(X \cdot i = Y \cdot i) \leq (X = Y)$

(P2) $c, d = c \sqcup d$

(P3) $\mathtt{func}(X, f, m) \sqcup \mathtt{func}(X, g, n) = \mathit{false}$ if $f/m \not\equiv g/n$

where $X$ and $Y$ are variables, $c \in \mathit{Con}$,
and $[Y/X]$ syntactically substitutes every $X$ by $Y$.

**Extended Axioms.**

(U4) $(X = f(t_1, \ldots, t_n)) = (f(t_1, \ldots, t_n) = X)$
$= \mathtt{func}(X, f, n) \sqcup \bigsqcup_{i=1}^{n} (X \cdot i = t_i)$

(U5) $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) = \bigsqcup_{i=1}^{n} (s_i = t_i)$

(U6) $f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n) = \mathit{false}$ if $f/m \not\equiv g/n$

where $X$ is a variable, and $s_i$ and $t_j$ are terms.

Fig. 4.2: Axioms for the Store.

root variables, among which is $\alpha$ that is used as the formal parameter in the language.

For any variables $X$ and $Y$, we assume the constraint $X = Y$ in $\mathit{Con}$ representing the *unification* between them.

For any variable $X$ and (function) symbol $f$ and non-negative integer $n$, we assume that there exists the constraint $\mathtt{func}(X, f, n)$ in $\mathit{Con}$ representing that the variable $X$ is bound to a *functor* '$f/n$'.

We also assume that for each variable $X$ there exists the hiding operator $\exists_X : \mathit{Con} \to \mathit{Con}$ that is meant to map any constraint to its partial constraint on the variables that are not captured by $X$. For example, we have $\exists_{\mathtt{X}}(\mathtt{Y = f(X, X \cdot 4)}) = \mathtt{func(Y, f, 2)} \sqcup (\mathtt{Y \cdot 1 \cdot 4 = Y \cdot 2})$ if $\mathtt{X}$ does not capture $\mathtt{Y}$.

Figure 4.2 summarizes the properties the store and certain constraints must enjoy. The semantics of the unification between terms is also defined there.

**Syntax.**

$$\textit{Agents} \quad A ::= \mathbf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i \mid A \parallel A \mid \mathbf{stop} \mid \exists X \langle A, c \rangle \mid p(X)$$

where $c \in \mathit{Con}$, $n \geq 1$, $p \in \mathit{Pred}$

and $X$ is a root variable other than $\alpha$.

**Transition Axioms.**

(R1) $\langle \mathbf{tell}(c), d \rangle \longrightarrow \langle \mathbf{stop}, c \sqcup d \rangle$

(R2) $\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, d \rangle \longrightarrow \langle A_j, d \rangle$ if $c_j \leq d$

(R3) $\langle p(X), d \rangle \longrightarrow \langle \exists \alpha \langle \overline{\exists} \alpha \mathit{Prog}(p),\ \alpha = X \rangle, d \rangle$

(R4) $$\frac{\langle A, c \sqcup \exists_X d \rangle \longrightarrow \langle A', c' \rangle}{\langle \exists X \langle A, c \rangle, d \rangle \longrightarrow \langle \exists X \langle A', c' \rangle, d \sqcup \exists_X c' \rangle}$$

(R5) $$\frac{\langle A, d \rangle \longrightarrow \langle A', d' \rangle}{\langle A \parallel B, d \rangle \longrightarrow \langle A' \parallel B, d' \rangle}$$

(R5′) $$\frac{\langle B, d \rangle \longrightarrow \langle B', d' \rangle}{\langle A \parallel B, d \rangle \longrightarrow \langle A \parallel B', d' \rangle}$$

Fig. 4.3: The Operational Semantics.

### 4.2.2 Agents

Given the set *Pred* of predicate symbols, we define the syntactic class of *agents* as in Figure 4.3.

The agent $\mathbf{tell}(c)$ adds constraint $c$ to the store, while $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$ nondeterministically waits for the store to entail some $c_i$ and then behaves as $A_i$. $A_1 \parallel A_2$ is parallel composition while **stop** is the terminated agent. $\exists X \langle A, c \rangle$ is the agent $A$ with an internal store $c$ that holds local information on $X$. $p(X)$ is a predicate call to $p$.

We abbreviate $\exists X \langle A, \mathit{true} \rangle$ to $\exists X A$. We denote by $\mathsf{Vars}(A)$ the set of root variables that capture some term contained in $A$, and abbreviate $\exists X_1 \ldots \exists X_n A$ to $\overline{\exists} X A$ if $\{X_1, \ldots, X_n\} = \mathsf{Vars}(A) \setminus \{X\}$. For a root variable $X$ that does not capture terms $t_1, \ldots, t_m$, we *term-abbreviate* $\exists X(\mathbf{tell}(X = \mathsf{a}(t_1, \ldots, t_n)) \parallel p(X))$ to $p(t_1, \ldots, t_n)$.

### 4.2.3 The Transition System

We define a *program* as a mapping $\mathit{Prog} : \mathit{Pred} \rightarrow \mathit{Agents}$. The code in Figure 4.1 can be thought of as a concrete representation of the program in Figure 4.4.

Given a program *Prog*, we define the operational semantics as the smallest set $\longrightarrow \subseteq (\mathit{Agents} \times \mathit{Con})^2$ that enjoys the axioms in Figure 4.3.

(R3) expresses the commencement of a goal $p(X)$ with a new store deployed. (R4) refreshes internal store as well as shared store for each reduction of the subagent.

### 4.2.4 The Observables

One can justify a compiler by showing that the object code it generates behaves as specified by the program. We will define the observables of a process by the phenomena we want to regard as its whole behavior.

The behavior of an agent $A$ can only be observed through computations beginning with $\langle A, \mathit{true}\rangle$ under a given program *Prog*. We therefore define for *Prog* the *observables* $\mathcal{O}[\![A]\!]_{\mathit{Prog}}$ of $A$ as follows:

$$\begin{aligned}\mathcal{O}[\![A]\!]_{\mathit{Prog}} \stackrel{\text{def}}{=} & \{c' \cdot \bot \mid \langle A, \mathit{true}\rangle \longrightarrow^* \langle A', c\rangle,\ c' \leq c\} \\ \cup\ & \{c \cdot \mathit{dd} \mid \langle A, \mathit{true}\rangle \longrightarrow^* \langle A', c\rangle \not\longrightarrow,\ A' \notin \mathbf{Stop}\} \\ \cup\ & \{c \cdot \mathit{tt} \mid \langle A, \mathit{true}\rangle \longrightarrow^* \langle A', c\rangle \not\longrightarrow,\ A' \in \mathbf{Stop}\} \\ \cup\ & \{c \cdot \infty \mid \langle A, \mathit{true}\rangle \longrightarrow^* \langle A', c\rangle \longrightarrow^\infty\}\end{aligned} \tag{4.1}$$

where $\langle A, c\rangle \longrightarrow^\infty$ means the existence of an infinite computation beginning with $\langle A, c\rangle$ (said $\langle A, c\rangle$ *diverges*) and **Stop** is the set of agents containing no asks, tells, or calls.

The $c \cdot \bot$ represents the possibility that the agent $A$ may cause the store to entail $c$. On the other hand, $c \cdot \mathit{dd}$, $c \cdot \mathit{tt}$, and $c \cdot \infty$ represent the possibility that $A$ may suspend, terminate, and diverge, respectively, if store reaches $c$.

It is known that collecting observables $\mathcal{O}[\![C[A]]\!]_{\mathit{Prog}}$ for all contexts $C[\cdot]$ generates a compositional denotational semantics of the agent $A$ under the program *Prog* [3, 8].

## 4.3 Interfaces

In this section we introduce *interfaces* of a process with which we formalize the behavior of a process under some class of input. Using interfaces, we can guarantee the correctness of intermediate code we generate.

Before formalization, we introduce some preliminaries.

### 4.3.1 Result of Local Choices

A compiler is allowed to resolve local choices statically, which exist within an agent as nondeterministic branches. We define a relation on agents to formalize such resolving.

Given a program *Prog*, we define that $A'$ is a *result of local choices* of $A$ if, for any context $C[\cdot]$, $\mathcal{O}[\![C[A]]\!]_{\mathit{Prog}} \supseteq \mathcal{O}[\![C[A']]\!]_{\mathit{Prog}}$ holds, in which case we will write $A \succeq A'$.

Note that the relation $\mathbf{ask}(c) \rightarrow A \succeq \mathbf{ask}(c) \rightarrow A'$ means that $A$ can be compiled into $A'$ under the store $c$. Thus, $\succeq$ can formalize the specialization of processes with nondeterminacy on scheduling as well as on local choices.

### 4.3.2 Upward-Closed Sets and Type Constructors

We say a set $S$ is an *upward-closed* set if $\forall c \in S \forall d \geq c\,(d \in S)$ holds. We will use an upward-closed set of constraints as an abstract store in the analysis. We typically use $\vec{c}, \vec{d}, \ldots$ to denote upward-closed sets of constraints.

Let us introduce a *type constructor*, an operator that maps a variable to an upward-closed set of constraints. For instance, assuming that $\mathtt{int}(X)$ is a constraint representing $X$ is bound to an integer, we define the type constructor $i$ by $i(X) \stackrel{\text{def}}{=} \uparrow \mathtt{int}(X)$ where $\uparrow c \stackrel{\text{def}}{=} \{d \in \mathit{Con} \mid d \geq c\}$. Likewise, we define $n(X) \stackrel{\text{def}}{=} \uparrow \mathtt{func}(X, [], 0)$ for *nil*, and $c(X) \stackrel{\text{def}}{=} \uparrow \mathtt{func}(X, ., 2)$ for *cons*.

We extend the hiding operator $\exists_X$ to an upward-closed set in a natural way:

$$\exists_X(\bigcup_{i \in I} \uparrow c_i) \stackrel{\text{def}}{=} \bigcup_{i \in I} \uparrow \exists_X c_i \tag{4.2}$$

For example, we have $\exists_X(i(X{\cdot}1) \cap c(\alpha{\cdot}2)) = c(\alpha{\cdot}2)$.

For root variables $X$ and $Y$, we define the operator $\{X \mapsto Y\}$ on upward-closed sets as follows:

$$(\bigcup_{i \in I} \uparrow c_i)\,\{X \mapsto Y\} \stackrel{\text{def}}{=} \bigcup_{i \in I} \uparrow ((\overline{\exists}_X c_i)[Y/X]) \tag{4.3}$$

where $\overline{\exists}_X(c) = \exists_{X_1} \ldots \exists_{X_n}(c)$ and $\{X_1, \ldots, X_n\}$ is the set of variables syntactically occurring in $c$ and not captured by $X$. This operator is used to represent the variable renaming on a predicate call. For example, we have $(i(X{\cdot}1) \cap c(\alpha{\cdot}2))\,\{X \mapsto \alpha\} = i(\alpha{\cdot}1)$.

### 4.3.3 Formalization of Interfaces

We formalize an interface of agents in terms of operational semantics as in Figure 4.5. Firstly, we define the syntactic class of *interfaces*.

The interface $\vec{c} \rightarrow \vec{d} \gg E$ represents the property that for any input store $c \in \vec{c}$ the agent can make the store evolve into $d \in \vec{d}$ without suspension and then behave as $E$ unless the agent diverges. In general, $\vec{c} \rightarrow \exists X_1 \ldots \exists X_k \sum_{j=1}^{m} (\vec{d_j} \gg E_j)$ represents the property that for any input store $c \in \vec{c}$ there exists some $j$ such that after declaring new variables $X_1, \ldots, X_k$ the agent can make the store evolve into some $d \in \vec{d_j}$ without suspension and then behave as $E_j$ unless the agent diverges.

The interface $A$ represents the property that the agent can behave as the agent $A$. $E_1 \wedge E_2$ represents the property that the agent can behave as $E_1$ and can behave as $E_2$. $E_1 \parallel E_2$ represents the property that the agent can behave as the parallel composition of the two agents each of which behaves as $E_1$ and $E_2$, respectively.

We then define the relation $\geq$ as in Figure 4.5. The definition (I5) makes $\geq$ a relation on interfaces that satisfies reflexive and transitive laws, which enables us to infer the properties of an interface, rather than of an agents.

**Syntax.**

*Interface* $E ::= \vec{c} \to \exists X_1 \ldots \exists X_k \sum_{j=1}^{m} (\vec{c_j} \gg E_j)$
$\mid A$
$\mid E \wedge E$
$\mid E \parallel E$

where $k \geq 0$, $m \geq 1$, $X$ is a root variable, $A$ an agent, and $\vec{c}$ is an upward-closed set of constraints.

**Inductive Definition.**

$\geq \subseteq \mathit{Agents} \times \mathit{Interface}$ ::

(I1) $A \geq B$ iff $A \succeq B$

(I2) $A \geq E_1 \wedge E_2$ iff $A \geq E_1$, $A \geq E_2$

(I3) $A \geq E_1 \parallel E_2$ iff $\exists B_1, B_2 \in \mathit{Agents}$ (
$A \succeq (B_1 \parallel B_2)$, $B_1 \geq E_1$, $B_2 \geq E_2$ )

(I4) $A \geq \vec{c} \to \exists X_1 \ldots \exists X_k \sum_{j=1}^{m} (\vec{d_j} \gg E_j)$ iff $\forall c \in \vec{c}$ (
$\langle A, c \rangle \longrightarrow^{\infty}$ or $\exists j \, \exists d \in \vec{d_j} \, \exists B \geq E_j$ ( $\mathbf{ask}(c) \to A$
$\succeq \mathbf{ask}(c) \to \exists X_1 \ldots \exists X_k (\mathbf{tell}(d) \parallel B)$ ))

$\geq \subseteq (\mathit{Interface} \setminus \mathit{Agents}) \times \mathit{Interface}$ ::

(I5) $E \geq E'$ iff $\forall A \in \mathit{Agents} ( A \geq E \Rightarrow A \geq E' )$

Fig. 4.5: Formal Definition of Interfaces.

## 4.4 The Interface Analysis

In this section, we explain our bottom-up method of analyzing sequentiality that uses a call graph of predicates. The analysis is formulated as the inference of interfaces.

In the analysis, we must analyze interfaces not of a goal but of the predicate itself. So, we abbreviate $\bar{\exists}\alpha \mathit{Prog}(p)$ to $p$ and analyze this agent. We assume that any variable $X \cdot i$ that syntactically occurs in such an agent is guarded by some $\mathbf{ask}(c)$ such that $\mathtt{func}(X, f, n) \leq c$ with some $f$ and $n \geq i$ if $X \not\equiv \alpha$. It is also assumed that any goal occurs in a term-abbreviated form with 'sufficient' arity.

### 4.4.1 Linear Interfaces

We call an interface of the form:

$$\bigwedge_{i \in I} (\vec{c_i} \to \exists X_0 \ldots \exists X_k \sum_{j \in J(i)} (\vec{d_{i,j}} \gg q_{i,j}(X_0))) \tag{4.4}$$

with $k \geq 0$ a *linear interface*. The union $\bigcup_{i \in I} \vec{c_i}$ is called its *input assumption*.

We assume $\mathit{Prog}(\mathtt{halt}) = \mathbf{stop}$ in order to express the termination, and we will abbreviate $\exists X_0 (\vec{d} \gg \mathtt{halt}(X_0))$ to $\vec{d}$. We call an interface of the form $\vec{c} \to \vec{d}$ a *sequential*

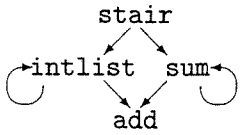



Fig. 4.6: A Call Graph.

*interface*. These two forms of interfaces are used in the bottom-up inference of those interfaces themselves.

Most of built-in predicates have their own sequential interfaces. For instance, `add` that performs integer addition enjoys $\texttt{add} \geq i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \rightarrow i(\alpha{\cdot}3)$.

The $q_{i,j}(X_0)$ is called a *tail call*. In reality, we must keep the original, term-abbreviated form for each tail call to perform the analysis.

### 4.4.2 Bottom-up Analysis of Predicates

The bottom-up interface analysis is performed in the following steps:

1. Build a call graph, namely, a directed graph whose nodes are predicates and whose arcs represent the caller-callee relationship between predicates.

2. Choose a node without outgoing arcs, remove it from the graph and try to find a linear interface of the corresponding predicate. This step is repeated until every node has outgoing arcs.

3. Choose a strongly connected component, which represents (mutually) recursive predicates, remove it from the graph and try to find their linear interfaces. Go back to step 2 if any node remains.

If interfaces of already removed predicates cannot be applied in the analysis either for inability of the analyzer or for deadlock of the program, we abandon the analysis and optimization of the predicate and of any predicate that may eventually call it. For these predicates, we will generate 'general' code that may spawn many goals.

**Example 1** The call graph of the program in Figure 4.4 is shown in Figure 4.6. We first remove the node `add` without any outgoing arcs. Since `add` is a built-in predicate, its interface is known prior to the analysis. We try to find interfaces of `intlist` and of `sum`, and then of `stair`.

### 4.4.3 Bottom-up Analysis of Agents

We describe how to compute a linear interface of an agent in a bottom-up manner. The analysis directs intermediate code generation that is explained in Section 4.5.

Inferring interfaces relies on abstract interpretation. We use an upward-closed set of constraints as an abstract store that represents (1) dynamic type information and (2) alias (variable-variable unification) information. The analysis requires linear interfaces, and therefore the dynamic type information including recursive data types as well, to be manipulated as data structures in a compiler, whose design is out of the scope of this paper.

Receiving a pair of an agent and its *initial input assumption* $\vec{a}$, the analysis recursively computes a linear interface for each subagent. The analysis may fail, which takes place in recursive calls (it becomes a tail call candidate) and other cases such as deadlocks. For parallel composition agents, the static scheduling information is also computed. The analysis is performed as follows:

$p(X)$: If a linear interface $\bigwedge_{i\in I}(\vec{c_i} \rightarrow \exists X_0 \ldots \exists X_k \sum_{j\in J(i)}(\vec{d_{i,j}} \gg q_{i,j}(X_0)))$ has been computed for $p$ where $X \not\equiv X_0$, return $\bigwedge_{i\in I}(\vec{c_i}\{\alpha \mapsto X\} \rightarrow \exists X_0 \ldots \exists X_k \sum_{j\in J(i)} \exists_\alpha(\exists_{X_1} \ldots \exists_{X_k} \vec{d_{i,j}} \cap \uparrow(\alpha = X)) \gg q_{i,j}(X_0)))$. Otherwise, we fail.

**tell**$(c)$: Same as built-in predicate calls explained above.

$\exists X\langle A, c\rangle$: We first compute a linear interface $\bigwedge_{i\in I}(\vec{c_i} \rightarrow \exists X_0 \ldots \exists X_k \sum_{j\in J(i)}(\vec{d_{i,j}} \gg q_{i,j}(X_0)))$ of $A$ with initial input assumption $\exists_X \vec{a}$. If $\exists_X \vec{c_i} = \vec{c_i}$ holds for each $i$, return $\bigwedge_{i\in I}(\vec{c_i} \rightarrow \exists X_0 \ldots \exists X_k \exists X \sum_{j\in J(i)}(\vec{d_{i,j}} \gg q_{i,j}(X_0)))$. Otherwise, we fail.

$\sum_{h\in H}$**ask**$(b_h) \rightarrow A_h$: If $\uparrow b_h \supseteq \vec{a}$ holds for some $h \in H$, let $H' = \{h\}$ so as to eliminate the local choice. Otherwise, let $H' = \{h \in H \mid \vec{a} \cap \uparrow b_h \neq \uparrow \mathit{false}\}$. Next, for each $h \in H'$ we compute a linear interface $\bigwedge_{i\in I(h)}(\vec{c_{h,i}} \rightarrow \exists X_0 \ldots \exists X_k \sum_{j\in J(h,i)}(\vec{d_{h,i,j}} \gg q_{h,i,j}(X_0)))$ of $A_h$ with initial input assumption $\vec{a} \cap \uparrow b_h$. Then return their conjunction (use $\bigwedge$). We fail if $H' = \{\}$.

$A_1 \parallel \ldots \parallel A_n$: We use abstract interpretation based on dynamic type information to schedule subagents.

The scheduling proceeds in the following steps:

1. Let the initial state of the abstract store be the initial input assumption $\vec{a}$.
2. For each subagent, try to find a linear interface and then a sequential interface.
3. Choose an agent with a sequential interface whose input assumption is entailed by the current abstract store. An interface that contains no alias information has the advantage. Remove the agent from the composition, and strengthen the abstract store by consulting the interface. This step is repeated as many times as applicable.
4. If no agent can be removed under the current abstract store, re-compute linear interfaces of the remaining agents with the current abstract store being initial input assumption.

5. If we have more than one agent with linear interfaces, we try to interleave them by the method explained later. If successful, we have an interleaved agent with a linear interface.
6. If we have exactly one agent with linear interfaces, try to find a sequential interface of it. If successful, remove the agent from the composition. The difference $\vec{d}$ of its input assumption $\vec{d} \cap \vec{c}$ from the current abstract store $\vec{c} \cap \vec{a}$ is lifted up to the input assumption of the whole parallel composition being analyzed. After this, the current abstract store is updated.
7. If exactly one term-abbreviated goal remains, choose it as the tail call. Otherwise, we fail.

Most agents without recursion or with a single tail-recursive call can be sequentialized with this algorithm. To deal with other forms of recursive predicates, other techniques such as conversion to tail-recursive forms should be used in conjunction [1].

Some subagents return alias information in output. By applying it to the remaining subagents, some variables may be represented by means of $\alpha$, thus possibly eliminating the number of variables used.

It should be straightforward to prove that every agent has the respective interface that is returned by the method.

**Example 2** We show the analysis for the agent

$$L \equiv \mathbf{tell}(\alpha{\cdot}3 = [\alpha{\cdot}1 \mid \mathtt{S}]) \parallel \mathtt{add}(\alpha{\cdot}1,1,\mathtt{K}) \parallel \mathtt{intlist}(\mathtt{K},\alpha{\cdot}2,\mathtt{S})$$

found in `intlist` with initial input assumption $i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2)$. Let us find an interface of $\mathtt{add}(\alpha{\cdot}1,1,\mathtt{K})$, namely $\exists \mathtt{G}(\mathbf{tell}(\mathtt{G} = \mathtt{a}(\alpha{\cdot}1,1,\mathtt{K})) \parallel \mathtt{add}(\mathtt{G}))$. We infer $\mathtt{add}(\mathtt{G}) \geq i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \rightarrow i(\mathtt{G}{\cdot}3)$ from the interface of `add`. Since $\mathbf{tell}(\mathtt{G}{=}\mathtt{a}(\alpha{\cdot}1,1,\mathtt{K})) \geq i(\alpha{\cdot}1) \rightarrow i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \cap \uparrow(\mathtt{G}{\cdot}3 = \mathtt{K})$, we have $(\mathbf{tell}(\mathtt{G} = \mathtt{a}(\alpha{\cdot}1,1,\mathtt{K})) \parallel \mathtt{add}(\mathtt{G})) \geq i(\alpha{\cdot}1) \rightarrow i(\mathtt{K}) \cap i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \cap \uparrow(\mathtt{G}{\cdot}3 = \mathtt{K})$, thus $\mathtt{add}(\alpha{\cdot}1,1,\mathtt{K}) \geq i(\alpha{\cdot}1) \rightarrow i(\mathtt{K})$. Seeing $\mathtt{add}(\alpha{\cdot}1,1,\mathtt{K})$ can be sequentially executed and contains no alias information, we schedule this first; and then $\mathbf{tell}(\alpha{\cdot}3 = [\alpha{\cdot}1 \mid \mathtt{S}])$, which has $i(\alpha{\cdot}1) \rightarrow c(\alpha{\cdot}3) \cap i(\alpha{\cdot}3{\cdot}1) \cap \uparrow(\mathtt{S} = \alpha{\cdot}3{\cdot}2)$. The abstract store is strengthened to $i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \cap c(\alpha{\cdot}3) \cap i(\alpha{\cdot}3{\cdot}1) \cap i(\mathtt{K}) \cap \uparrow(\mathtt{S} = \alpha{\cdot}3{\cdot}2)$ and the alias information is applied to the tail call. Now, it is easy to infer the relation $L \geq i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \rightarrow \exists \mathtt{G}(\ c(\alpha{\cdot}3) \cap i(\alpha{\cdot}3{\cdot}1) \cap i(\mathtt{K}) \cap i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \cap \uparrow(\mathtt{S} = \alpha{\cdot}3{\cdot}2) \cap \uparrow(\mathtt{G}{\cdot}3 = \alpha{\cdot}3{\cdot}2) \gg \mathtt{intlist}(\mathtt{G})\ )$.

### 4.4.4 Inferring Sequential Interfaces

We mention how to infer sequential interfaces from linear interfaces. Essentially, it consists of two steps:

1. find an input sufficient to execute the process without suspension, and

2. compute an output it can perform on termination.

These procedures are performed by computing fixed points on dynamic type information for each strongly connected component removed from the call graph.

### 4.4.5 Process Interleaving

We must also consider a method of interleaving parallel composition, and here is how to utilize the interface information to interleave producer and consumer processes. The method is basically unfold/fold transformation [11] but is directed by the interfaces of the two processes, which enables us to justify sequentializing specialization. We explain only the simplest case, but it gives us a good insight into our analysis.

1. Let $P$ and $Q$ be processes to be interleaved which have tail calls to $p$ and $q$, respectively. We assume the given linear interface of $p$ (and $q$, respectively) has a single tail call to $p$ (and $q$) which was $P'$ (and $Q'$) in the original, term-abbreviated form.

2. Build a list that enumerates the variables occurring in the recursive calls $P'$ and $Q'$.

3. For each shared variable between $P$ and $Q$, unite the elements of the list that correspond to the paths within which the shared variable is passed.

4. Introduce new predicates $p'$ and $q'$ such that their formal arguments are specified by the above list and that $P'$ and $Q'$ can be represented as calls to $p'$ and $q'$, respectively.

5. Introduce a new predicate $r = p' \parallel q'$ and confirm that $P \parallel Q$ can be represented as a call to $r$.

6. Compute linear interfaces of $p'$ and $q'$ from those of $p$ and $q$, and then try to find a linear interface of $r$, using the definition of $r$ to obtain a tail call to $r$.

**Example 3** Let us interleave the two predicate calls in

$$\texttt{intlist(1,}\alpha\texttt{·1,S)} \parallel \texttt{sum(S,}\alpha\texttt{·2,}\alpha\texttt{·3)}$$

in `stair`.

Since `S` is a shared variable, the third argument of `intlist` and the first argument of `sum` are unified. Accordingly, we introduce $\texttt{intlist}' = \texttt{intlist(}\alpha\texttt{·1,}\alpha\texttt{·2,}\alpha\texttt{·3)}$, $\texttt{sum}' = \texttt{sum(}\alpha\texttt{·3,}\alpha\texttt{·4,}\alpha\texttt{·5)}$ and $r = \texttt{intlist}' \parallel \texttt{sum}'$. We can infer that

$$\texttt{intlist(1,}\alpha\texttt{·1,S)} \parallel \texttt{sum(S,}\alpha\texttt{·2,}\alpha\texttt{·3)} \succeq r\texttt{(1,}\alpha\texttt{·1,S,}\alpha\texttt{·2,}\alpha\texttt{·3)}.$$

Thus, we go on to find a linear interface of $r$ from the linear interfaces of `intlist` and `sum`:

$$\begin{aligned} r \geq i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \cap i(\alpha{\cdot}4) \rightarrow \exists \mathtt{G}( \\ ( n(\alpha{\cdot}3) \cap i(\alpha{\cdot}5) \gg \mathtt{halt(G)} ) \\ + ( c(\alpha{\cdot}3) \cap i(\alpha{\cdot}3{\cdot}1) \cap i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \cap i(\mathtt{G}{\cdot}4) \\ \cap \uparrow(\mathtt{G}{\cdot}3 = \alpha{\cdot}3{\cdot}2) \cap \uparrow(\mathtt{G}{\cdot}5 = \alpha{\cdot}5) \gg r(\mathtt{G}) )) \end{aligned} \tag{4.5}$$

## 4.5 Code Generation

In this section, we define an intermediate language and explain the code generation in our framework.

### 4.5.1 Definition of Intermediate Code

Figure 4.7 summarizes our low-level, sequential intermediate code. This subsection can be skipped in the first reading, for it is not the subject of the paper.

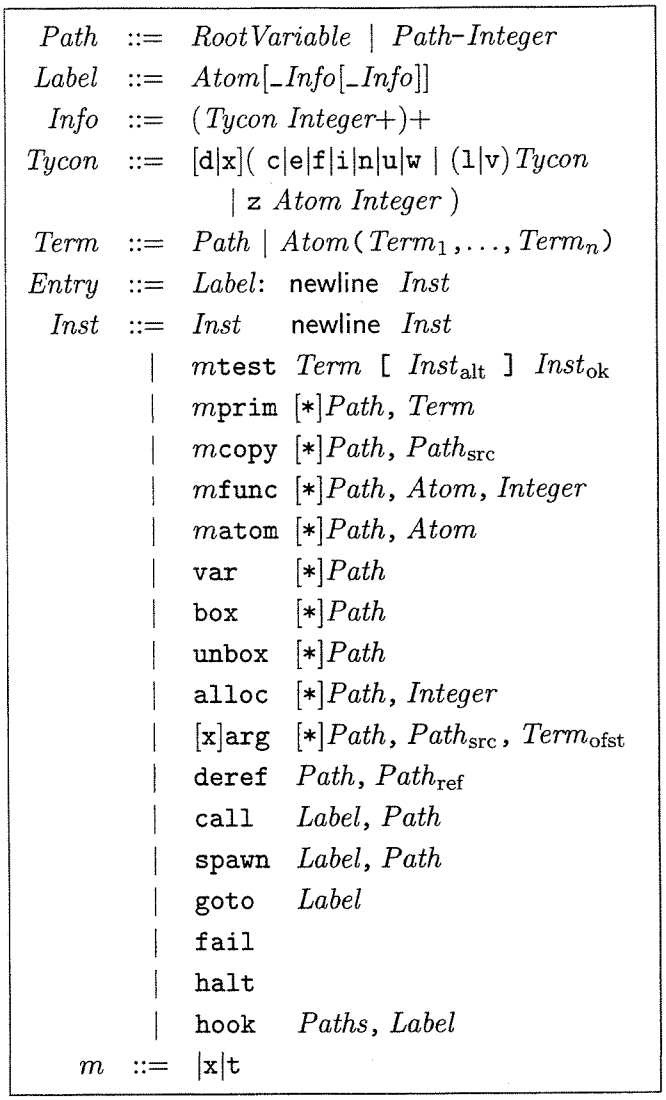
The code explicitly manipulates memory through variables, and the memory consists of cells. For explanation, we assume a pointer-tag implementation, that is, each cell is tagged with one of FUNC, FUNCREF, or REF. Each variable is implemented by a cell, and is a *path*. The path $X$-$i$ refers to the cell containing the $i$-th element (offset $i+1$) of the structure referenced by the path $X$.

The instruction "`var` $X$" acquires a new cell that has a reference to itself with REF tag, and assigns it to the location specified by $X$. The instruction "`alloc` $X$,$n$" acquires $n+1$ contiguous, uninitialized cells and assigns the reference to its first cell with FUNCREF tag to the location specified by $X$.

The instruction "`copy` $X$,$Y$" assigns the content of the location specified by $Y$ to the location specified by $X$. The instruction "`func` $X$,$f$,$n$" assigns the atomic value '$f/n$' with FUNC tag to the location specified by $X$. The "`atom` *Path*,*Atom*" is shorthand for "`func` *Path*,*Atom*,`0`". The instruction modifier `t` means that the consistency of the output with the current content of the destination path is recursively checked. The recursive case happens only in `tcopy`, which exactly performs general unification.

A label *p_Encode*($\vec{a}$) describes the entry point of the predicate $p$ with the input assumption $\vec{a}$, which must be entailed by the store when the code led by the label is executed. We assume *Encode* generates an *Info* that describes the dynamic type information on $\alpha$. For example, the code implementing the built-in predicate `add` can have the input assumption $i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2)$ and is led by the label `add_i1i2`. In this paper, we assume that the input assumption specified in the label argument of an instruction is statically guaranteed to be entailed. The definition details of *Label* are not the subject of the paper.

The variable `A` is used as the actual argument register throughout the code and always



| | | |
|---|---|---|
| *Path* | ::= | *RootVariable* \| *Path-Integer* |
| *Label* | ::= | *Atom*[_*Info*[_*Info*]] |
| *Info* | ::= | (*Tycon Integer*+)+ |
| *Tycon* | ::= | [d\|x]( c\|e\|f\|i\|n\|u\|w \| (l\|v) *Tycon* |
| | | \| z *Atom Integer* ) |
| *Term* | ::= | *Path* \| *Atom*( $Term_1, \ldots, Term_n$ ) |
| *Entry* | ::= | *Label*: newline *Inst* |
| *Inst* | ::= | *Inst* newline *Inst* |
| | \| | mtest *Term* [ $Inst_{alt}$ ] $Inst_{ok}$ |
| | \| | mprim [*]*Path*, *Term* |
| | \| | mcopy [*]*Path*, $Path_{src}$ |
| | \| | mfunc [*]*Path*, *Atom*, *Integer* |
| | \| | matom [*]*Path*, *Atom* |
| | \| | var [*]*Path* |
| | \| | box [*]*Path* |
| | \| | unbox [*]*Path* |
| | \| | alloc [*]*Path*, *Integer* |
| | \| | [x]arg [*]*Path*, $Path_{src}$, $Term_{ofst}$ |
| | \| | deref *Path*, $Path_{ref}$ |
| | \| | call *Label*, *Path* |
| | \| | spawn *Label*, *Path* |
| | \| | goto *Label* |
| | \| | fail |
| | \| | halt |
| | \| | hook *Paths*, *Label* |
| *m* | ::= | \|x\|t |

Fig. 4.7: An Intermediate Language.

replaces $\alpha$. The content of A is assumed to be a reference to a non-shared structure (i.e., referenced by only one cell) with *sufficient* arity for executing the code. Any object referenced by A-$i$ may be shared unless the corresponding path has the type modifier d in the label. The scopes of variables other than A are confined between two labels.

The intermediate code is executed sequentially until it reaches one of goto, halt, fail, or hook instructions. "spawn $l$,$G$" enqueues to the *process pool* a new goal goal($l$,$G$) whose entry point is $l$ and whose argument is $G$. "hook *PathList*,$l$" hooks goal($l$,A) to the set of paths in *PathList*. Every tell involving some path in *PathList* causes the hooked goal to be spawned to the process pool. "goto $l$" jumps to the label $l$. "call $l$,$G$" is a non-suspending built-in call to the label $l$.

The test instruction tests entailment of the specified constraint. If the entailment

is not yet observed, the alternative code specified in the argument is executed.

The "`arg` $X,Y,K$" assigns the reference to the $K$-th argument of the structure pointed to by $Y$ to the location specified by $X$. The `deref` fully dereferences a chain of REF pointers. The * that prefixes a destination path specifies that the destination path is dereferenced by one level. These enable the uninitialized variable optimization.

The instructions "`unbox` $X$" and "`box` $X$" convert the content of $X$ to its unboxed and boxed value, respectively. Once unboxed, any `arg`, `copy`, `prim` and `test` instructions must be prefixed with the instruction modifier `x`, and the type modifier `x` must be used in the corresponding path.

The formal semantics of the code, which is required to justify code optimization formally, is omitted in the paper.

### 4.5.2 Code Generation Directed by Interface Analysis

Let $\vec{a}$ be the input assumption of the given linear interface. The code generation is performed as follows:

$p(X)$: If we have found a sequential interface of $p$ whose input assumption is entailed by $\vec{a}$, the code is inline expanded. Built-in predicates fall under this case. It is desired to choose an interface with more assumption and more output. Return the code with `A` replaced by $X$ and other variable names replaced by fresh variable names. Inline expansion may cause code explosion, but in most cases each copied code would be compacted through specialization.

If we have found no such interfaces, return `spawn` $p\_Encode(\vec{a}\,\{X \mapsto \alpha\}),X$. If the goal is a tail call, it may be immediately rewritten to `copy` `A`$,X$ followed by `goto` $p\_Encode(\vec{a}\,\{X \mapsto \alpha\})$.

**tell**$(c)$: Same as built-in predicate calls explained above.

For **tell**$(X = Y)$ where $\uparrow$`func(`$Y,f,0$`)` $\supseteq \vec{a}$, return `tatom` $Lookup(X),f$. For **tell**$(X = Y)$ of other cases, return `tcopy` $Lookup(X),Lookup(Y)$.

For **tell**$(X = f(t_1,\ldots,t_n))$, return `alloc` $W,n$; `func` $W$`-0`$,f,n$; code for **tell**$(W \cdot 1 = t_1)$; ...; code for **tell**$(W \cdot n = t_n)$; and `tcopy` $X,W$ where $W$ is a fresh root variable.

We assume *Lookup* translates any variable to a path.

$\exists X\langle A, c\rangle$: For $c \neq true$, return the code for the agent $\exists X\langle$**tell**$(c) \parallel A, true\rangle$. For $c = true$, firstly build the code for $A$ preceeded by `new` $X$. Then return the code with all $X$ replaced by a fresh variable name.

$\sum_{h=1}^{m}$**ask**$(c_h) \rightarrow A_h$: Choose an $h$ according to the interface. Return a `test` instruction that performs entailment check of $c_h$ (a part of which may be statically done using $\vec{a}$) followed by the code for $A_h$. It contains the code for the remaining branches

```
intlist_i1i2:
  var S
  var K
  test ilt(A-1,A-2) [
    tatom A-3,[]
    halt
  ]
  tprim K,iadd(A-1,1)
  alloc G,2
  func  G-0,.,2
  copy  G-1,A-1
  var   G-2
  tcopy A-3,G
  alloc H,3
  copy  H-1,K
  copy  H-2,A-2
  copy  H-3,A-3-2
  copy  A,H
  goto  intlist_i1i2
```

(a) Before Optimization

```
intlist_i1i2:
  test ilt(A-1,A-2) [
    tatom A-3,[]
    halt
  ]
  alloc G,2
  func  G-0,.,2
  copy  G-1,A-1
  var   G-2
  tcopy A-3,G
  prim  A-1,iadd(A-1,1)
  copy  A-3,A-3-2
  goto  intlist_i1i2
```

(b) After Optimization

Fig. 4.8: Generated Intermediate Code.

as its alternative code argument. If the check is found to pass always, the `test` instruction (as well as the code for the rest of branches) may be omitted.

$A_1 \parallel \ldots \parallel A_n$: Return the code for the subagents concatenated in the scheduled order.

For efficiency reasons, we should, in fact, use a list of unconstrained variables in code generation.

Figure 4.8 (a) shows the generated code of `intlist`.

## 4.6 Code Optimization

We will briefly mention the optimization on the sequential intermediate code though it is not the subject of this paper.

Optimization in our framework consists of two stages: one is to generate sequential intermediate code, and the other is to optimize the generated code. The former is primarily concerned with process scheduling (and local choice elimination) for the avoidance of suspension and process interleaving, and is directed by the sequentiality analysis. The latter relates to both the code level optimization such as copy propagation, and the implementation-level details including tag elimination and update-in-place. Our framework can straightforwardly justify the safety of such low-level optimization based on the intermediate language.

We should mention that other analysis frameworks that justify particular optimization techniques can be peacefully incorporated into our framework to build a more powerful and efficient compiler. For instance, the linearity analysis [10] can be incorporated so as to statically guarantee the safety of update-in-place optimization applied to global data, and mode analysis [12] can statically determine which interface to be used for each unification.

**Example 4** Figure 4.8 (a) is optimized into Figure 4.8 (b). Figure 4.9 (b) shows the optimized code of `stair`, while its *general* (i.e., unspecialized) code can be like Figure 4.9 (a). We can see that the sequentializing specialization has significant importance. Further low-level optimization, such as tag elimination, can be applied but is not shown because it is out of the scope of this paper.

## 4.7 Related Work

P. Van Roy [13] demonstrates an optimizing compilation framework for Prolog that uses low-level intermediate language, rather than WAM (Warren's Abstract Machine), to make code specialization more effective. The optimization techniques explained there can be systematically applied to concurrent logic programs only if sequentiality is extracted from the code. Our sequentiality analysis contributes to the sequentialization phase in the optimizing compilation framework for concurrent logic languages (and other fine-grained concurrent languages to some extent).

Related work on optimizing compilers for fine-grained concurrent languages include [4, 5].

Debray [4] describes a sequentializing compiler for the concurrent constraint language Janus. It shares many concepts with this paper including the extraction of sequentiality using modular boundness analysis. However, their compiler uses Prolog as the sequential intermediate code, and memory management optimization is completely delegated to that of the underlying Prolog compiler. It means that the implementation techniques unique to fine-grained concurrent language that require for instance the analysis of interleaving cannot be achieved.

Debray et al. [5] explain top-down non-suspension analysis and its application to an optimizing compiler for Janus that uses C as its sequential intermediate code. Fixed point dataflow analysis, however, becomes complicated in the presence of complex message flow, which is the *raison d'être* of concurrent logic programming. We believe that bottom-up analysis will do the job with better precision and modularity, and will smoothly connect the extracted sequentiality to the runtime system.

```
stair:
  alloc G,3
  var   G-1
  copy  G-2,A-2
  copy  G-3,A-3
  spawn sum,G
  copy  A-2,A-1
  copy  A-3,G-1
  atom  A-1,1
  goto  intlist

sum:
  test wait(A-1) [
    hook [A-1],sum
  ]
  test func(A-1,.,2)[
    test func(A-1,[],0)[
      fail
    ]
    tcopy A-3,A-2
    halt
  ]
  alloc G,3
  copy  G-1,A-1-2
  var   G-2
  copy  G-3,A-3
  spawn sum,G
  copy  A11,A-1-1
  copy  A-1,A-2
  copy  A-2,A11
  copy  A-3,G-2
  goto  add
```

(a) General Code

```
stair_i1i2:
  copy A-4,A-2
  copy A-2,A-1
  copy A-5,A-3
  var  A-3
  atom A-1,1
  goto r_i1i2i4

r_i1i2i4:
  test ilt(A-1,A-2) [
    tatom A-3,[]
    tcopy A-5,A-4
    halt
  ]
  prim A-4,iadd(A-4,A-1)
  var  A-3
  prim A-1,iadd(A-1,1)
  goto r_i1i2i4
```

(b) Specialized Code

Fig. 4.9: General Code vs. Specialized Code.

## 4.8 Conclusion and Future Work

We have presented a framework for extracting sequentiality in concurrent logic programs and for generating its sequential intermediate code. The proposed framework is based on bottom-up analysis using interfaces that formalize certain sequential properties of a process, and hence can justify the intermediate code it generates.

Future work includes a definition of the formal semantics of the intermediate language, which enables us to guarantee the correctness of code optimization. It is also important to implement and evaluate an optimizing compiler for concurrent logic programs written in Flat GHC.

## References


[1] A. W. Appel, Compiling with Continuations, Cambridge University Press, 1992.

[2] M. Codish, M. Falaschi and K. Marriot, "Suspension Analyses for Concurrent Logic Programs", ACM TOPLAS, Vol. 16, No. 3, 1994, pp. 649–686.

[3] F. S. de Boer and C. Palamidessi, "On the Semantics of Concurrent Constraint Programming", Proc. ALPUK 92, Workshops in Computing, Springer-Verlag, 1992, pp. 145–173.

[4] S. Debray, "QD-Janus: A Sequential Implementation of Janus in Prolog", Software Practice and Experience, Vol. 23, No. 12, 1993, pp. 1337–1360.

[5] S. Debray, D. Gudeman and P. Bigot, "Detection and Optimization of Suspension-Free Logic Programs", J. of Logic Programming, Vol. 29, Nos. 1–3, 1996, pp. 171–194.

[6] M. Falaschi, P. Hicks and W. Winsborough, "Demand Transformation Analysis for Concurrent Constraint Programs", Proc. 1996 Joint Int. Conf. and Symp. on Logic Programming (JICSLP'96), The MIT Press, 1996, pp. 333–347.

[7] A. King and P. Soper, "Schedule analysis of concurrent logic programs" Joint Int. Conf. and Symp. on Logic Programming, The MIT Press, 1992, pp. 478–492.

[8] V. A. Saraswat, M. C. Rinard and P. Panangaden, "Semantic Foundations of Concurrent Constraint Programming", Proc. of POPL'91, ACM Press, 1991, pp. 333–352.

[9] K. Ueda, "Concurrent Logic/Constraint Programming: The Next 10 Years", The Logic Programming Paradigm: A 25-Year Perspective, K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren (eds.), Springer-Verlag, 1999, pp. 53–71.

[10] K. Ueda, "Linearity Analysis of Concurrent Logic Programs", Proc. Int. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications, Ito, T. and Yuasa, T. (eds.), World Scientific, 2000, pp. 253–270.

[11] K. Ueda and K. Furukawa, "Transformation Rules for GHC Programs", Proc. Int. Conf. on Fifth Generation Computer Systems 1988, ICOT, 1988, pp. 582–591.

[12] K. Ueda and M. Morita, "Moded Flat GHC and Its Message Oriented Implementation Technique", New Generation Computing, Vol. 11, No. 3–4, 1993, pp. 3–43.

[13] P. Van Roy, Can Logic Programming Execute as Fast as Imperative Programming?, Ph. D. thesis, University of California, 1990. Available as Report No. UCB/CSD 90/600.

# 第5章　KL1分散処理系DKLICの設計と実装

多くの分散プログラムでは, アプリケーションの本質部分よりも, 分散化のために必要となる同期などのマルチプロセスに関する処理や, 動的ネットワークに対応するための処理の記述に多くの労力を消費していた. 一方でGHC (Guarded Horn Clauses) に基づく並行論理型言語KL1では, 通信を論理変数を使って表現し, 同期処理を簡潔に記述することができる.

本研究では, このKL1の処理系KLICを拡張し, 分散処理記述を容易にするネーミングサービス, コード移送, プロセス移送, 遠隔述語呼出などの機能も提供できる処理系DKLICの実装を目指した. そのために, 論理変数による通信路をTCPソケットで実装し, その最適化のためのプロトコルの設計と部分的な実装を行った. これに加え, 分散論理変数に対するネーミング機能をPeer to Peerベースで実装することで, ネットワーク透過な環境の提供と動的なDKLICアプリケーションネットワーク構築を可能にした. 結果的に, 分散アプリケーション記述を大幅に簡単化するような広域分散プログラミング環境の提供に成功した.

## 5.1　はじめに

### 5.1.1　背景

多くの分散アプリケーションは, JAVAやC++などの手続き型言語によって開発されてきた. しかし, もともと逐次処理を前提に作られたためにマルチプロセス間通信の記述などが容易ではなかった. これを解決するような広域分散プログラミング環境を提供するために並列論理型言語KL1の分散処理系が提案 [1] され, Javaによって実装されたKL1処理系KL1j[1] と, [2, 3, 4] などに代表されるようなUnix上のKL1処理系であるKLICをKL1とC言語によって分散拡張した処理系という2系統が存在した. 後者の系統の処理系では, 通信を論理変数で表現できるKL1の特性をTCPのソケット上に実装することで高水準な2者通信である分散論理変数を実現していた. また, 複数のノードがアドホックに連係するような分散アプリケーションのために, ネットワーク透過な環境を提供するネーミングモジュール [5, 6] も提案されていた. しかし, 分散論理変数モジュールは分散論理変数によるネットワークの接続状況を把握ができず, 通信路の最適化を制御することができなかった. また, ネーミングモジュールは接続状況を把握していたが, 分散論理変数モジュールとは独立して実装されていたために同様に通信路の最適化を行なえず, また, 動的にネットワーク接続を拡大する機能がなかった.

本論文では, これらの独立したKLIC分散拡張モジュールを協調させた新たなKL1分

散処理系 DKLIC を提案し, ネーミングモジュール, 最適化プロトコルの設計とその一部の実装し, その実装をに基づいて分散論理変数を再実装した. これにより, Peer to Peer ベースの動的ネットワークの構築とネットワーク透過な広域分散プログラミング環境を実現した. また, これまでに存在した遠隔述語呼出 [3], ゴールやプロセスの移送 [7] といった分散処理特有の機能の位置づけを再検討した.

### 5.1.2 本論文の構成

本論文は, まず 5.2 節では KL1 について述べる. 5.3 節では DKLIC の仕様と設計について述べる. DKLIC は大まかに分散論理変数層とネーミング層の 2 層構成である. 5.4 節で分散論理変数層の仕様を述べる. 5.5 節で分散論理変数層の設計について述べる. この節では実装に先だって問題となった KLIC の内部機構や実装の方針について論じている. 5.6 節で分散論理変数層の設計と実装について述べる. 設計において明らかにした問題とその解決法に基づいている. 5.7 節ではネーミング層の仕様と設計について, 5.8 節でネーミング層について述べる. 5.9 節で実装したプログラムについての評価をし, 5.10 節で関連研究について述べ, 最後に 5.11 節でまとめと今後の課題について論じる.

## 5.2 並行論理型言語 KL1

この節では KL1 についての概要を述べた後, 例題を挙げてプログラムとその実行の様子について簡単に説明する.

### 5.2.1 KL1 の概要

KL1 は Flat GHC に基づいた記号処理を並列に処理するための論理型言語である. 同種の言語としては Concurrent Prolog, Parlog, Fleng などがある.

KL1 は項書き換え系言語であり, そのプログラムは以下のような形式の節 (ルール) の集合とモジュールの宣言によって成り立ってる。

$$h :- G \mid B.$$

$h$, $G$, $B$ はヘッド, ガード, ボディと呼ばれ, 全てゴール列である. ただし, h は単一ゴールで構成される. ゴールは以下のような形式である.

述語名 (引数 1, 引数 2, ...)

KL1 プログラムの実行は, Prolog などの項書き換え系言語とは大きく異なる. それは, 書き換えを待つゴールがスタックではなくランダムに取り出せるゴール集合である点と, 複数のゴールに対する書き換えを並列に行える点である. また, 書き換えのときに引数が完全に具体化している必要がない. ただし, ヘッドとのマッチの他にガードに書き換え条件を付記できる. これらの特徴によって並列処理を簡潔に記述することができる. つまり, ゴール集合中のゴールは並行プロセスと見なせ, 複数のゴール間で共有する変数は並行プロセス間の通信路と見なせる.

```
:-module main.
main:-  intlist(1,10,List),
        sum(List,0,Res).
sum(List,Sm,Res):- List=[]          | Res = Sm.
sum(List,Sm,Res):- List=[X|List2]  |
    Sm2 := X + Sm,   sum(List2,Sm2,Res).
intlist(N,Max,List):- N > Max | List = [].
intlist(N,Max,List):- N =<Max |
    List = [N|List2],   N1 := N + 1,
    intlist(N1,Max,List2).
```

図 5.1: KL1 のプログラム例

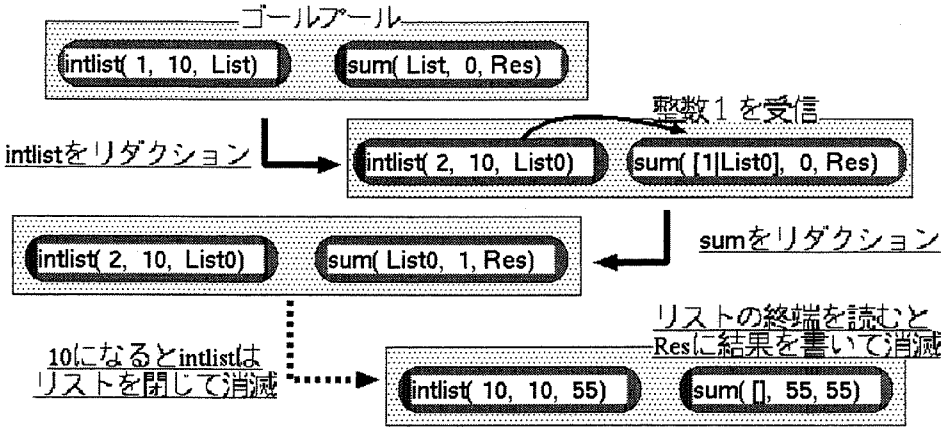



図 5.2: ゴールプールの遷移

### 5.2.2 サンプルプログラムとその実行について

図 5.1 が KL1 のサンプルプログラムであり, 図 5.2 がその実行途中のゴールプールの概念図である. まず処理系が main という初期ゴールをゴールプールに投入することで実行が開始する. まず, main ゴールは, intlist(1,10,List) と sum(List,Res) というゴールに書き換えられる. これらのゴールは状態を変化させながら再帰的に書き換えられていくためプロセスと言える. intlist プロセスは, 1 から 10 までの整数を変数 List にリスト要素として徐々に具体化していく. sum プロセスは intlist プロセスが具体化していくリスト要素を全て加算していき, リストを終端まで読むと変数 Res に結果を具体化する. ただし, sum プロセスはこのリストが具体化されるまでは書き換えをせずに待つ. このような待ち合わせ機能は 2 プロセス間の同期通信を可能にする. また, 前節で述べたように, 変数 List のような 2 つのプロセス間で共有する遅延具体化ストリームはプロセス間の通信路と見なせ, 特にチャネルと呼ぶことがある.

このように KL1 が並列処理, マルチプロセス間通信や同期処理などを簡潔に記述でき

```
/** intlist/3提供側 **/
:-module main.
main :-  ns:register([name(intlist_3)],Ls),
         Ls = [listen(normal(S)],
 S = (A,B,C),
         intlist(A,B,C).
intlist(N,Max,List):- N > Max | List = [].
intlist(N,Max,List):- N =<Max |
      List = [N|List2],  N1 := N + 1,
      intlist(N1,Max,List2).

/** intlist/3利用側 **/
:-module main.
main :-  ns:lookup([name(intlist_3)],normal(S)),
         S = (1,10,List),
         sum(List,0,Res).
sum(List,Sm,Res):- List=[]          | Res = Sm.
sum(List,Sm,Res):- List=[X|List2] |
      Sm2 := X + Sm,  sum(List2,Sm2,Res).
```

図 5.3: DKLIC のプログラム例 1

ることから, 本研究では分散処理記述に KL1 を採用し, その処理系 DKLIC の実装を目指した. 以降の節ではこの DKLIC について述べていく.

## 5.3 DKLIC について

### 5.3.1 DKLIC のプログラム例

DKLIC を用いたプログラム例を図 5.3 に示す. これは 5.2 節の図 5.1 を DKLIC で書いた場合である. 各述語定義部分は全く同じだが, これらは TCP ソケットを用いたサーバクライアントプログラムになっている. この例では intlist/3 側は接続毎に立ち上げなくてはならない. サーバクライアント型のアプリケーションにするならば, intlist/3 側を図 5.4 のようにすればよい.

### 5.3.2 DKLIC の想定する分散環境

DKLIC が想定する分散環境とは, IP アドレスを持ったマシン (ホストと呼ぶ) が動的に増減し, 各ホストのアーキテクチャが不均一であるような環境を指す. つまり, あるホストで実行可能なコードが他ホストで実行可能かどうか分からないような環境である. ま

```
:-module main.
main :-  ns:register([name(intlist_3)],Ls),
         Ls = [listen(Res)|Ls2],
         server(Res,Ls2).
server(Res,Ls) :- Res=normal(S) |  S = (A,B,C),
      intlist(A,B,C),  % intlist 呼び出し部分
      Ls = [listen(Res2)|Ls2],
      server(Res2,Ls2).
intlist(N,Max,List):- N > Max | List = [].
intlist(N,Max,List):- N =<Max |
      List = [N|List2],  N1 := N + 1,
      intlist(N1,Max,List2).
```

図 5.4: DKLIC のプログラム例 2

た, 各ホストは複数の DKLIC アプリケーション (ノード) を持つことができる. ノードはいつネットワークから切断されるか予想できず, 新たなノードが接続してくるかも予想できない. ノードも同様である. また, 特定のノードにアクセス可能かどうかを事前に知ることはできない.

### 5.3.3 DKLIC の構成

DKLIC の階層図を図 5.5 に示す. 本論文では DKLIC コア部分を実装した.

**分散論理変数層** ネットワーク上の 2 ノード間にまたがる論理変数 (分散論理変数) を提供する. KL1 のチャネルには具体化された項だけでなく, 未定義変数を含む項を具体化することができる. DKLIC ではこの特徴も分散論理変数に持たせている. また, 不要な中継者を排除する通信路の最適化機能も持つ. 図 5.3 のサンプルプログラムにおいては論理変数 `Ls` をトップとする論理変数が分散論理変数である.

ネーミング層より上のアプリケーションは, 分散論理変数を他の変数と区別することなく使える. しかし, 使用するには図 5.3 の `ns:lookup` や `ns:register` といったネーミング層の API を通さなくてはならない. いいかえると, これらの API の引数に現れる未定義な論理変数 (及びそれをトップレベルとするような論理変数) が分散論理変数である.

このような隠蔽を行なったのは, ネーミング層の実装にポートディスパッチャの役割を持たせるためである. 通信路最適化機能ではクライアントプログラムであっても TCP のソケットを bind しなくてはならない. 複数のノードがホスト内に乱立して無秩序にポートを消費するのを抑える必要が生じたためである.

**ネーミング層** DKLIC では, ネーミング層によってネットワーク透過な環境を提供する. 狭義にはネーミングとは分散論理変数に対する名前付けとその解決を意味しているが, あ

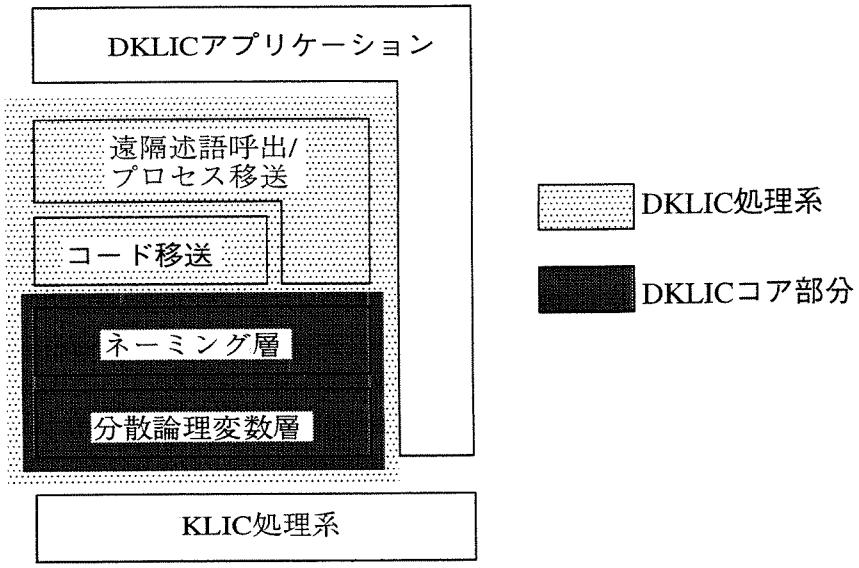



図 5.5: DKLIC の階層図

るサーバへのチャネルをサービス名で名前付けると見なした場合サービス検索という意味にもなる.

重要な点は, ネーミング層の実装であるネーミングプロセスは, 前述のようにポートディスパッチャでもあるので, 1 ホストに 1 プロセスだけ起動していなければならない点である. また, このプロセスは動的に他のネーミングプロセスへ接続し, 複数のネーミングプロセスが協調してチャネルの名前付け解決を行なう. これによってネーミングプロセスは P2P ベースの DKLIC ネットワークを構築する.

図 5.6 は, 図 5.3 のサンプルプログラムが接続している DKLIC ネットワークの一例である. DKLIC ネットワークはネーミング層より上の階層から見るとブラックボックスになっている. 唯一確かなのは, `intlist/3` や `sum/3` などの DKLIC アプリケーションの接続先が, そのアプリケーションが動いているホスト内のネーミングプロセスであるということだけである.

**コード移送** 上位のゴール移送や遠隔述語呼出が必要とする層である. ゴールの移送先で, そのゴールのルールが存在しない場合に用いられる. コードとは, 人が読む事ができるような, KL1 のプログラムかよりコンパクトな中間コードを指す.

**ゴール移送** 主にエージェントなどのアプリケーションを記述するときに必要となる. ゴールを移送し, 移送先で実行可能になるように準備する. [7] では, 中間コードとして [8] を採用したコード移送と共にゴール移送を実装しているが, 中間コードへのコンパイラなどは未実装である.

**遠隔述語呼出** リモートプロセジャーコールとほぼ同義である. この層も含めたコード移送から上の層はネーミングの対象とする. つまり, コード移送サーバ, ゴール移送サー

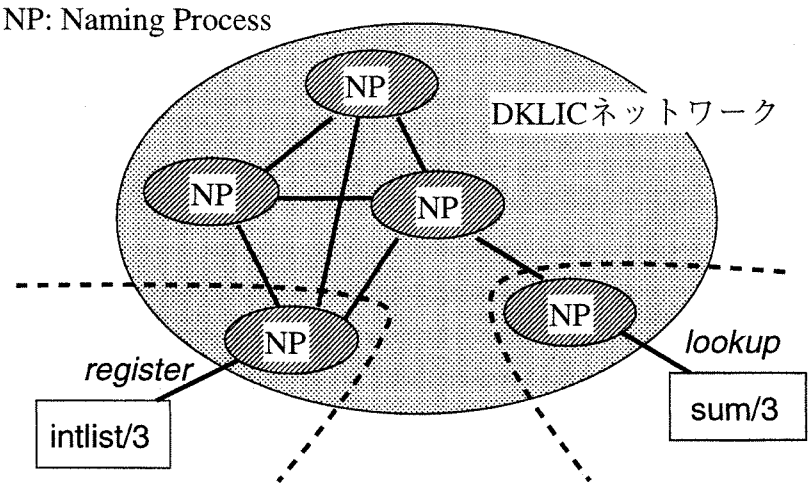



図 5.6: DKLIC ネットワーク

バ, 遠隔述語呼出サーバなどと呼ぶ. これによって各ノードが全ての機能を持つ必要がなくなり, 必要に応じてネットワーク上の各サーバを利用することができる.

次章から本論文で実装した分散論理変数層とネーミング層について述べる.

## 5.4 分散論理変数層の仕様

この層に求められる機能は, 分散論理変数の生成, 具体化, 廃棄というライフサイクルの管理と, 3 者以上で通信した場合に生じる中継者を排除する機能である. 本節ではこれらを解決する分散論理変数プロトコルと中継排除プロトコルの設計について述べる.

### 5.4.1 分散論理変数

分散論理変数プロトコルは [4] で示されているが, 中継排除プロトコルの理解に必要なため詳述する.

分散論理変数には 2 ノード間の KLIC 変数を同一視するための ID を割り当てる. 各サイトではこの ID と KLIC 変数の対応表を管理する KL1 プロセス (変数表プロセスと呼ぶ) を置く. 各ノードで起きた分散論理変数の生成, 具体化, 廃棄といったイベントは ID を用いて表現され, もう一方のノードへと通知され, 処理される. イベント, イベント処理, 及び ID の定義を行い, プロトコルを示す.

#### 分散論理変数のイベント

分散論理変数の状態変化にともなうイベントについて述べる.

**生成**　生成とは, 新たな分散論理変数が生成され新規にID割り当てが行なわれたことを意味する. 割当を行なったノードは, もう一方のノードへIDを用いてopenメッセージによって通知する. メッセージ受信側では, IDに対応するKLIC変数を新規に生成する. 当然だが生成メッセージの発行元では未使用のIDを割り当てる. しかし, 生成メッセージのほとんどは, 具体化メッセージ中に暗黙的に含ませることが可能である.

**具体化**　分散論理変数の具体化を意味する. 分散論理変数に対応するローカルなKLIC変数の具体化を感知したなら, IDと具体化項のセットをもう一方のノードへbindメッセージによって通知する. 具体化項に分散論理変数が含まれるなら, それらもIDで表現する. メッセージ受信側ではIDに対応するKLIC変数を受信した具体化項によって具体化する. ここで未知のIDが含まれていたなら新規にKLIC変数を作成する. これが前述の暗黙的な生成メッセージである.

**廃棄**　分散論理変数の廃棄を意味する. 元来KL1ではその処理系にGC実装をすることを前提としている. 従ってこの廃棄メッセージとその処理は, 分散論理変数におけるID空間のGCである. このID空間GCの詳細は後述する.

KL1変数は単一代入変数なため, 不要な分散論理変数とは具体化済み分散論理変数を意味する. 廃棄メッセージも具体化メッセージに暗黙的に含まれる. しかし, 実際のプロトコルでは, 中継排除プロトコルにおいて真に廃棄するまで具体化履歴を保存しなくてはならないため, closeとclosedメッセージによって廃棄を行う.

### ID

IDは, 分散論理変数の状態変化通知のために必要な整数ID部と, 2者通信そのものをグローバルに識別するためのコネクションID部のセットで表現する. コネクションIDは中継排除のためだけに必要である. ここでは, 整数ID部の設計についてのみ述べる. これは前述のID空間の効率の良いGCのための設計である.

前述の廃棄メッセージとその確認によってGCは可能だが, 通信する両者が使用するID空間を分けることでプロトコル一部省略できることがある. ID空間が分かれていないとノードBからopen(ID)メッセージが来るかもしれないので, ノードAからbindメッセージの直後にopen(ID)メッセージを送ることはできない (図5.7).

しかし, ホストAが生成のときに用いるID空間とホストBのID空間が別の場合 (図5.8). ホストBがそのIDで生成メッセージを送ってこないことが保証されているため, ホストAは具体化メッセージ送信と同時にそのIDを使用することができる. そこで, 2ホスト間でTCPコネクションを開設要請した側がゼロから始まる偶数のID空間を持ち, 要請された側が1から始まる奇数のID空間を持つこととした.

### 分散論理変数プロトコル

前述のイベント, イベント通知, ID空間をプロトコルにまとめる.

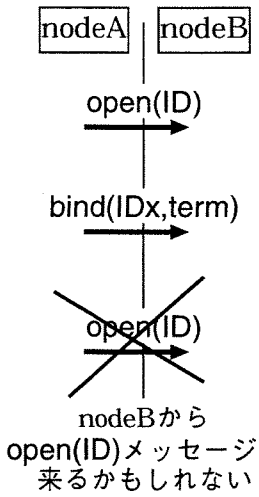



図 5.7: ID 空間分割しない場合

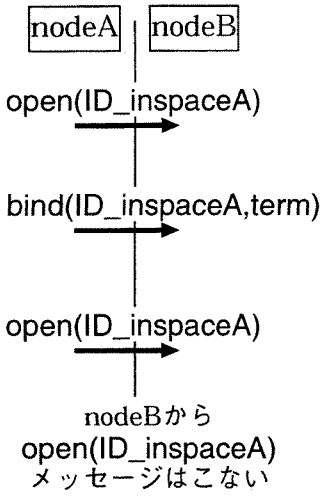



図 5.8: ID 空間分割した場合

1. TCP コネクションを開設要求した側はゼロから始まる偶数 ID 空間を持ち, 開設要求された側は, 1 から始まる奇数 ID 空間を持つ.

2. 次に, 開設した側がトップレベルの分散論理変数の `open` メッセージを発行する. 以降は, `bind` メッセージに未使用の ID を発見したなら, `open` メッセージが暗黙的に含まれていると解釈する.

3. 分散論理変数がユーザによって具体化された場合, ID と具体化項を引数にして `bind` メッセージを送信する. 受信側は ID に対応する KLIC 変数を具体化する.

4. 具体化メッセージの送信側 (ホスト A) がこの分散論理変の生成者でもあるなら, ホスト A 側でこの分散論理変数を廃棄完了待ち状態 (`closing`) にし, 相手 (ホスト B) からの廃棄完了 (`closed`) メッセージによって廃棄する.

   ホスト A が生成者でない場合は, ホスト A 側は廃棄完了待ち状態にし, ホスト B 側からの廃棄完了メッセージを待って, 廃棄する.

5. ある分散論理変数を廃棄するときに, この TCP コネクションにおいて未廃棄な分散論理変数がなかったなら (定数オーダーで調べられる), 通信終了メッセージを送信する. 相手からも終了メッセージの確認応答があったなら, TCP コネクションを閉じる.

図 5.3 のプログラムにおける分散論理変数の具体化操作を図 5.9 に示した.

### 5.4.2 中継排除

分散論理変数プロトコルは 2 者通信では十分であるが, 3 者通信のときに発生する中継者を排除することができない. 図 5.10 のように, ノード B で分散論理変数 X, Y を単一化

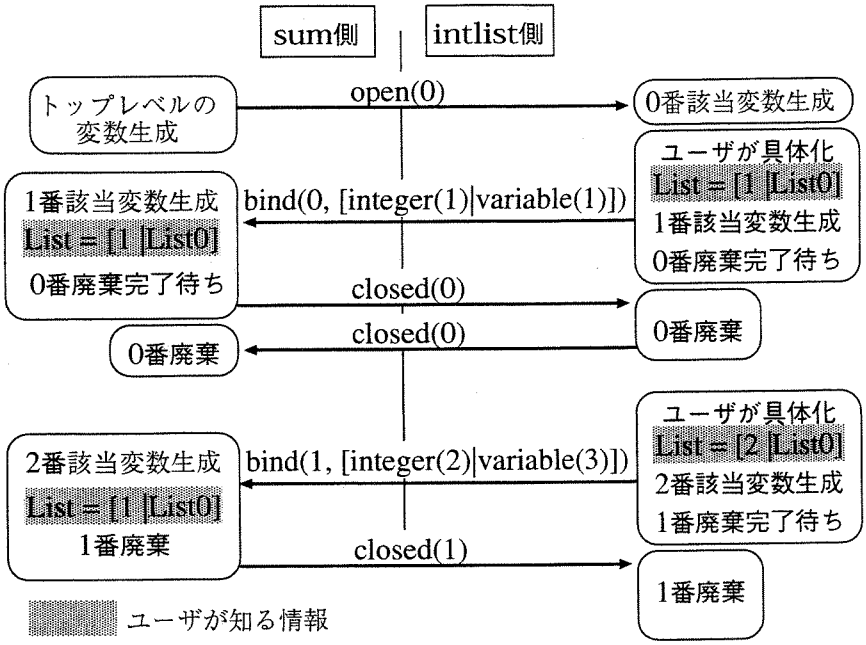



図 5.9: 分散論理変数プロトコルの流れ

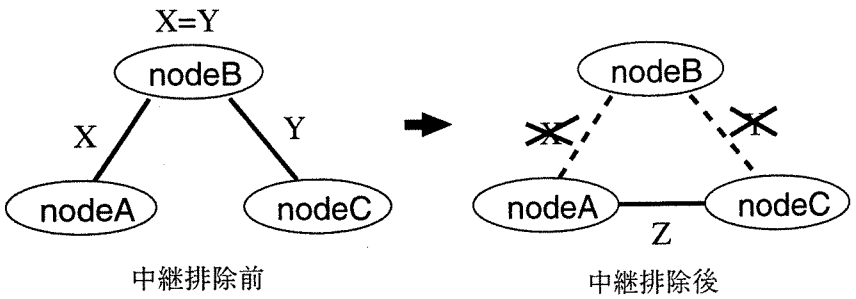



図 5.10: 中継者の排除

し, かつノードBにおいて変数X, Yを共に必要としない (参照しない) ならば, ノードBは, 一方の分散論理変数の具体化を読みとり, 他方の分散論理変数に具体化するだけの中継者となる. このようなノードの存在は, オーバーヘッドやセキュリティにおけるリスクを伴う. そこで, 中継排除プロトコルを設計した.

### 中継排除プロトコル

KLIC の共有メモリによる並列実装においても同様の中継排除 [9] が行なわれているが, [9] は, 生産者が消費者に対して、完全に具体化した要素のリストを送信する one-way message プロトコルである. 本研究では, 要素に未定義変数を含む構造体を持つことを考慮したプロトコルである. 本節ではこのプロトコルについて述べる. 詳細は [10] にも述べているが, 分散論理変数層の実装にあたって重要である.

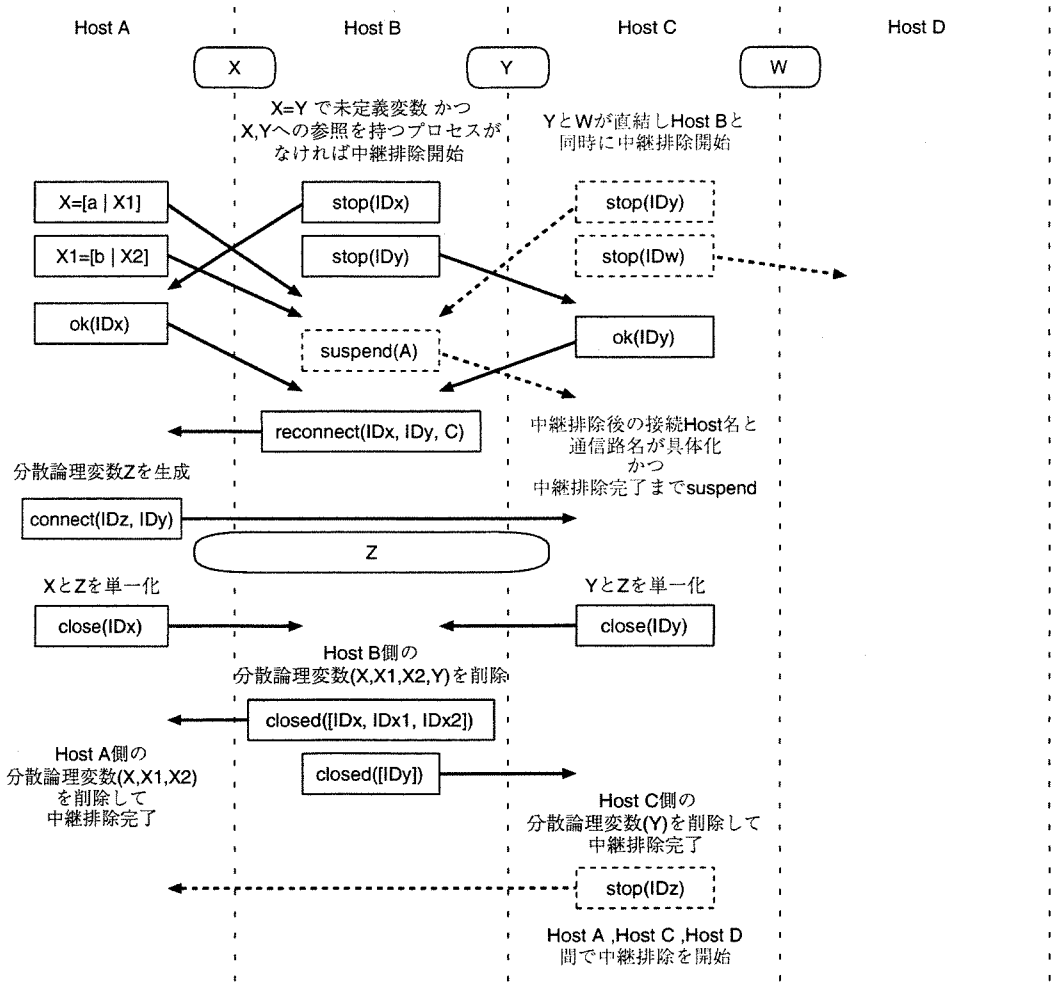



図 5.11: 中継排除プロトコル図

### 中継排除プロトコル

中継排除を実現するプロトコルは図 5.11 に示す．中継排除プロトコルの流れには，他のホストの中継排除と衝突が起きる場合と起きない場合に分類できる．

1. `stop` メッセージは，ホスト A とホスト C に送信される．引数が指す分散論理変数とその分散論理変数から派生する分散論理変数の具体化メッセージの送信をブロックさせる (他の変数はブロックしない) メッセージである．送信と同時に自らもブロックする．

   以降ではホスト A とホスト C に対して同じ作業を行なうので，ホスト A についてのみ記述する．

2. ホスト A が `stop` メッセージを受信し，送信停止処理が完了すると `ok` メッセージをホスト B に返信する．

ホストBでは, 送信したstopメッセージと行き違いにstopしたはずの変数の具体化メッセージがホストAから送られてくる場合がある. この具体化項に新規の分散論理変数が含まれていた場合, これもstopする. このために, ホストBでは, stopメッセージを送信後からokメッセージを受信するまでに受信した, 新規の分散論理変数のIDを記憶しておく. ここで記憶したIDは, 後に分散論理変数を削除する際 (5) に利用する. 前節で述べた廃棄完了待ち状態は, このバッファに記憶されている状態でもある.

3. ホストAとホストCからokメッセージを受信すると、reconnectメッセージを片方のホスト (この場合はホストA) に送信する。

4. reconnectメッセージを受信したホストAは, 指定されたホスト (ホストC) に対して新しいTCPコネクションを開く (接続済みならばそれを用いる). さらに, 新しい分散論理変数Zを生成し, ホストCに対して分散論理変数YとZを単一化するように要請するメッセージconnectを送信する。

5. ホストAはconnectを送ったら, 中継排除対象変数のトップレベルの変数 (この場合はX) と新規作成した変数Zとを単一化する. これにより, 中継排除プロトコル開始直後にホストBに送信してしまっていた具体化メッセージをホストCに対して再送信することができる. さらに, ホストBに対して変数Xの廃棄要請メッセージ (close) を送信する.

   ホストCはconnectを送られたら, YとZを単一化する. さらにホストBに対して変数Yの廃棄要請メッセージ (close) を送信する.

6. closeメッセージを受信したホスト (ホストB) は, 引数が示す分散論理変数とそこから派生した分散論理変数を廃棄する. 廃棄後に, 廃棄した分散論理変数のIDをclosedメッセージとして返信する. このclosedメッセージは, 分散論理変数プロトコルにおけるclosedメッセージと同じレベルのメッセージである (簡略化のためにリストになっている).

7. closedメッセージを受信したホスト (ホストA, ホストC) が, 引数が示す分散論理変数を削除したら, 中継排除は終了となる.

次に、衝突の起きる場合の、例外的な動作を赤い枠と線で表す。

X=YによりホストBが中継排除され, 同時にY=WによりホストCが中継排除されるとき, ホストBC間を同時にstop命令が送受信されることがある. この場合, 互いに通信相手が中継排除されることで, 通信路が破壊されることになる.

通信路の破壊を防ぐためには, 中継排除を行うホストに優先順位をつける必要がある. すべてのTCPコネクションには, connectしたホストとacceptしたホストの2種類が存在する. この場合, BC間コネクションは, ホストBがconnectしたものとする。

1. 同じTCPコネクションに対してstopメッセージが行き交った場合には, connectした側 (ホストB) の中継排除を優先することで通信路の破壊を防ぐ.

   もし, 中継排除が進行した状態 (okメッセージの受信以降) でstopメッセージを受け取った場合には, 進行中の中継排除を優先する.

2. 中継排除を優先して行うホストBは, 中継排除を待つホストCに対して, suspend メッセージを送信する. suspend メッセージは引数にはホストAの名前を与える. これは, 中継排除を待つホスト (ホストC) は, 再開後にはホストBCD間ではなく, ホストACDにおけると中継排除を行なう必要があるためである. ホストBからホストCへの再開メッセージは, ホストABCにおける中継排除完了をホストCが検知することである.

3. suspend メッセージを受け取ったホストは, 新しい分散論理変数 (変数Z) のIDが具体化し, 優先ホストの中継排除が終了まで中継排除を停止する. 中継排除を再開するときは、新しいホスト (ホストA) に改めて stop メッセージを送信する。

## 5.5 分散論理変数層の設計

分散論理変数層の実装は変数表プロセスと呼び, 上田高山両氏による中継排除を伴わない実装が存在した. 前節の分散論理変数プロトコルはこの実装で用いられていたプロトコルを一部変更したものである. しかし, 実装においては, 中継排除を導入することでいくつかの問題が生じた. 本節では, この問題とその解決法について論じ, それを元に新たな変数表プロセスの設計を行なう.

### 5.5.1 中継排除実装の問題点

5.4.2で述べたが, 中継排除するためには, (i) 分散論理変数同士がユニフィケーションしたことを検知し, (ii) それらにユーザからの参照がないことを確認しなくてはならない. しかし, この二つはKLIC上のKL1プログラムでは記述することができない. さらに (iii) 中継排除の時には複数の接続に属するID空間を扱うために, これらの接続を識別するためのコネクションID空間が必要になる. これら3つの問題の解決方法について述べる.

#### 分散論理変数同士の単一化検知

以下に示す節では, check(X,X) というゴールであっても, 2引数がともに両方具体化しないと限りリダクションされない. このためKL1プログラムから分散論理変数同士の単一化検知ができない.

```
check(A,A) :- true | true.
```

あるいは

```
check(A,B) :- A=B | true.
```

そこで分散論理変数同士の単一化を検知する方法が2通り考えられた. 一つは [4] で示された分散論理変数をジェネレータオブジェクトで実装しC言語による単一化関数を単一化のタイミングにフックする方法.

もう一つは分散論理変数のC言語レベルの同一性検査によって, 事後に単一化があったことを検出する方法である.

前者は採用できないことが判明し, 後者はKL1による記述は不可能だが, C言語関数での記述は可能であることが分かった. 以下には消去法によりC言語関数での分散論理変数同士の同一性検査を採用した過程を述べる.

#### Generatorオブジェクトによる実装の放棄

KLICでは開発者が新たに3種類のジェネリックオブジェクト (Data, Consumer, Generatorの3種) という型を定義が可能で, 必要な値や関数をC言語で実装すればよい. これらの関数はKLIC実行中に呼び出される.

唯一Generatorオブジェクトだけは, 単一化関数を定義可能である. この関数中で, 単一化の相手が分散論理変数であるかを知る事ができる. 図5.12は, KLICランタイムシステムのGeneratorオブジェクトの単一化関数の呼出部分である.

ランタイムはGeneratorオブジェクトと未定義変数 (あるいはGeneratorオブジェクト同士) の単一化のときに, オブジェクトの`generate`関数を呼出し, 生成された具体項で単一化相手を具体化する. しかし, この分散論理変数オブジェクトは複数回の単一化を検知するために, 具体項ではなく自分自身を返す`generate`関数を実装する必要がある. ところが, 図5.12を見るとGeneratorオブジェクトが自分自身を返した場合は, 再度ユニフィケーションを試行して, 無限ループを生成することが分かった. 従ってこの実装は採用できないことが明らかになった.

#### 分散論理変数同士の同一性検査

KLICの内部データ表現では, KLIC変数はポインタであり, 2つのKLIC変数の同一性はポインタ値の同一性に置き換えられる. そこで, 本実装では全ての分散論理変数を管理する変数表オブジェクト (Dataオブジェクト) を実装し, あるタイミングで管理下の同じポインタ値を持つ分散論理変数が存在するかを検査するC言語関数をフックする手法を採用した.

ここで検出した同一の分散論理変数はユーザ参照チェックにより, 参照が無いことを確認した上で, IDの組で表現してKL1プログラムへ通知し, 中継排除プロトコル処理に利用する. この関数を呼び出すタイミングはユーザ参照チェックの方法に依存しているため, KLICのGCのタイミングである. 詳しくは次で述べる.

#### ユーザ参照チェック

#### ユーザ参照チェックの方法

分散論理変数へのユーザ参照とは, DKLICシステム以外のゴールによる分散論理変数への参照のことである. そこで, まず変数表オブジェクトが管理するKLIC変数へのゴールからの参照をチェックする方法について述べる.

KLIC変数には参照数情報がない. したがって, 参照の有無を直接調べる方法は存在しない. ランタイムに調べる唯一の方法は, チェック対象変数がKLICのGC (コピーGC) 直後に新旧どちらの領域に存在するかで判断する方法である.

```
generator_unify(gsx, sy, allocp)
/*
   sx is a suspension structure
   and y is hook or generator
*/
{
      /************省略*************/
      } else {
        /* x and y are both generator,
   but both failed. */

        struct generator_object *gobjx =
                      untag_generator_susp(gsx->u.o);

        /****** generate メソッドを呼ぶが,
                この Generator オブジェクトは自身を返す. *******/

        q tmpx = generic_generate(gobjx, allocp);

        switch((long)tmpx) {
        case (long)makeref(0): /******* 省略 *******/
        case (long)makecons(0):  /******* 省略 *******/
        default:
          allocp = heapp;
          gsx->backpt = tmpx;
          if(isref(tmpx) && tmpx == derefone(tmpx)) {
            derefone(sy->backpt) = tmpx;
          } else {
  /**** この Generator オブジェクトの場合ここに来る ****/

          /**** この do_unify 呼び出しは, 結果的に全く同じ引数での
                    generator_unify(本関数) の呼び出しとなる.
            従って無限ループとなる ***/

                    return do_unify(allocp, tmpx, sy->backpt);
          }
      /************省略*************/
}
```

図 5.12: Generator オブジェクトを含む場合の KLIC の単一化処理関数 (unify.c)

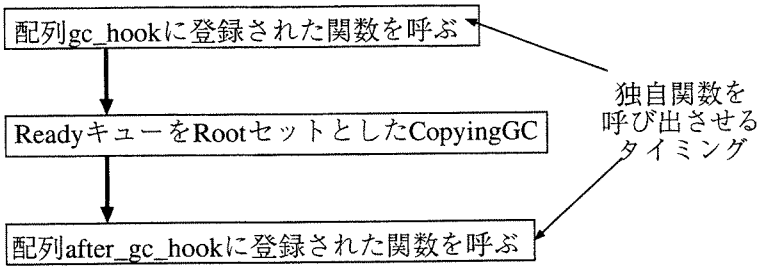



図 5.13: KLIC の GC の流れ 1

このコピー GC は Ready キューを Root セットとしている．そのためある変数が新領域に存在することは，その変数が Ready キューのゴールから到達可能であり，その到達経路上のゴールが参照していることを意味する．

また，前述の DKLIC システムのゴールとは，分散論理変数の具体化監視するゴールである．このゴールは性質上必ずチェック対象変数を中断原因変数として中断する．また，中断原因変数にフックしたゴールは，容易にそのゴール名 (実際には述語構造体変数へのポインタ) を識別できる．従って，ユーザ参照が無いことの必要十分条件は，以下のようになる．

1. KLIC の Copying GC が終了した直後に，
2. チェック対象変数が旧領域に存在し，かつ
3. 中断原因変数になっているが，
4. フックしているゴールが具体化監視ゴールしかない．

この判定を行なう関数は，GC の前後の KLIC による関数フックを用いて実装すると容易だが，使用できないことが明らかになる．しかし，この関数フックと類似の方法で解決するため，ここでは，KLIC の GC の流れ (図 5.13) と KLIC ソース (図 5.14) を示す．`after_gc_hook` や `gc_hook` が関数フックである．

### 参照ゴールの排除

変数表プロセスの実装では，常に Ready キューに存在するようなゴールから分散論理変数を直接間接を問わず参照してはならないことが明らかである．また，フックするゴールであっても数を極力抑えた方が検出が容易である．

一方で変数表プロセスは，リモートホストから送られてきた具体化メッセージによって自ホスト内の KLIC 変数を具体化する．一般にある変数を具体化するようなゴールは，ほとんどが Ready キュー中に存在するか，そこから辿れるゴールである．これは前述の条件判定法が無効であることを意味する．そこで，Data オブジェクトに具体化を行なわせる方法を採用した．

```
static struct goalrec *collect_garbage(qp)
     struct goalrec *qp;
{
  /** 省略 **/

  /**** Ready キューを Root セットとする Copying GC ****/
  for(; pq->prio >= 0; pq = pq->next) {
    pq->q = copy_one_queue(pq->q,allocp,ntop,otop,nsize,osize);
    allocp = heapp;
  }
  qp = copy_one_queue(qp, allocp, ntop, otop, nsize, osize);

  /** 省略 **/

  /****** この配列に関数ポインタを代入する API は
          alloc.c ファイルに公開されている ******/
  for (k=0; k<num_after_gc_hooks; k++) {
    heapp = after_gc_hook_table[k](heapp);
  }

  /** 省略 **/
}
```

図 5.14: KLIC ランタイムの GC 関数 (in gc.c)

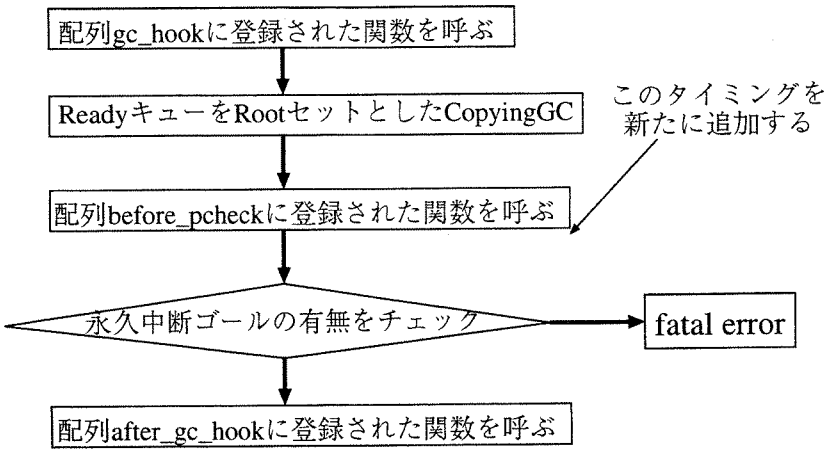



図 5.15: KLIC の GC の流れ 2

### GC 時の関数フックの新設

分散論理変数の GC 時のコピーをいつ行なうかが問題になる．前述のように Ready キューからのコピーが全て終った後でないと，参照チェックを行なえない．このタイミングは `after_gc_hook` であるように見える．

一方で，ジェネリックオブジェクトは自分自身がどのようにコピーされるかを独自に定義できるため，管理する変数が KLIC の GC によって勝手にコピーされることはない．ただし，管理する変数にフックするゴールをオブジェクトがコピーしなければならない．コピーしないと，具体化するゴールが存在しない変数にフックしている (永久中断している) と見なされ，処理系が異常終了する．

ここでジレンマが生じる．図 5.15 は永久中断ゴール検出も表示した GC の流れ図である．`after_gc_hook` で判定した後にコピーしなくてはならないが，そこでコピーすると永久中断ゴール検出に間に合わない．

そこで，KLIC を改造して新たに図 5.15 に示すような永久中断ゴールチェックの直前，Copying GC の直後に関数呼び出しを行なえるようにした．

### コネクション ID の設計

従来の P2P 通信のみを想定した実装では分散論理変数には整数 ID を割り当てていた．しかし，中継排除を行なう時には，“ホスト A との接続の 0 番とホスト B との接続の 0 番にあたる変数が単一化されている”，ということを表現しなくてはならない．さらに，リモートホストからの TCP コネクションは IP と 1 対 1 対応するが，ローカルホストから接続は複数あり，IP (文字列 localhost) では識別できない．そこで，コネクション ID は，コネクションの両端のノードの IP アドレスを元に生成し，ローカルホストからの接続には整数を一意に割り当てることとした．

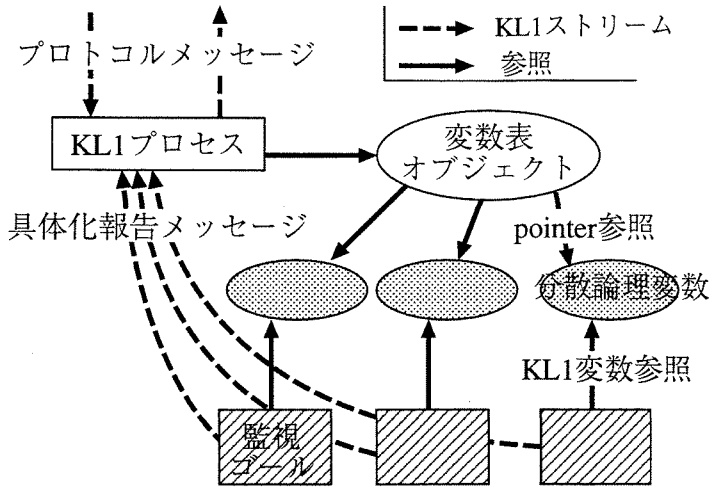



図 5.16: 変数表プロセスの構成図

### 5.5.2 変数表プロセスの全景

これまでに述べたことをまとめると以下のようになる.

- 分散論理変数同士の単一化は, ポインタの同一性検査によって検出する.
- ユーザ参照が無いことは, KLIC の Copying GC 直後にその変数が旧領域にあり, かつその変数には DKLIC システムのゴールしかフックしていない, ことで判断する. これは新規作成した `before_pcheck` フックで行なう.
- 分散論理変数の具体化監視ゴール以外で, 変数表プロセスのゴールが分散論理変数を参照するのを避けるために Data オブジェクトを実装する.
- ユーザ参照の有無をチェックするための関数呼び出しタイミングが無いので KLIC に新たなタイミングを追加する.
- コネクション ID は, コネクションの両端のノードの IP アドレスを元に生成し, ローカルホストからのコネクションには整数を一意に割り当てる.

これを満たし, さらに実装を容易にするために, 変数表プロセスは (i) 各プロトコルのメッセージ送受信を行なう部分を KL1 で, (ii) 分散論理変数への具体化や状態を管理する部分は Data オブジェクトで実装する. ただし, 具体化監視ゴールだけは KL1 で記述する. 各部分は図 5.16 のような関係になる.

## 5.6 分散論理変数層の実装

本節では, ID の実装, プロトコル処理を行なう KL1 部分, 分散論理変数を管理する変数表オブジェクトの実装について述べる.

### 5.6.1 IDs

ID は以下のような形式にした.

@(整数, コネクション ID)

コネクション ID は localhost との接続では整数を一意に割り当て, 遠隔ホストとの接続では, 両者の IP アドレスを文字列として連結し, それをアトムに変換したものを用いる. 例えば, 123.456.789.000 と 456.000.123.789 という IP を持つノード間のコネクションならば, 123_456_789_000_456_000_123_789 という ID になる. 文字列のままでは変数表オブジェクトの実装が複雑になるためにアトムに変換した. また, ピリオドは節の終端記号であり, ソケット通信において parse に失敗するので, アンダーバーに変換した.

### 5.6.2 分散論理変数プロトコル処理

プロトコル処理部分は以下のような KL1 プロセスによって構成した.

- 入力処理プロセス (in)
- 出力処理プロセス (out)
- 変数表オブジェクト保持プロセス (vart)
- 分散論理変数監視プロセス (observe)

各プロセスの関係を図 5.17 に示す.

#### in プロセス

このプロセスは, ソケットから読み込んだメッセージを処理するプロセスである. 受け付けるメッセージは, open, close, closed, finish, bind である.

**open(ID)** TCP コネクションが開設された時に最初に送られてくるメッセージ. これを受け取ったら vart プロセスに対して new_connection メッセージを発行し, 新規の接続を報告する.

**close(ID)** ID で示す分散論理変数の廃棄を要請するメッセージ. これを受け取ったら vart プロセスに対して, set_state(ID,closing) か set_state(ID,closed) メッセージを発行する. どちらになるかは, 分散論理変数プロトコルに従って決定する.

**closed(ID)** ID で示す分散論理変数の廃棄完了メッセージ. これを受け取ったら vart プロセスに対して, set_state(ID,closed) メッセージを発行する. 同時に channels メッセージを発行し, このコネクション中に存在する分散論理変数の総数を得る. この総数がゼロならば, out プロセスに対して finish メッセージを送る. このメッセージはそのまま, リモートノードへと送られる.

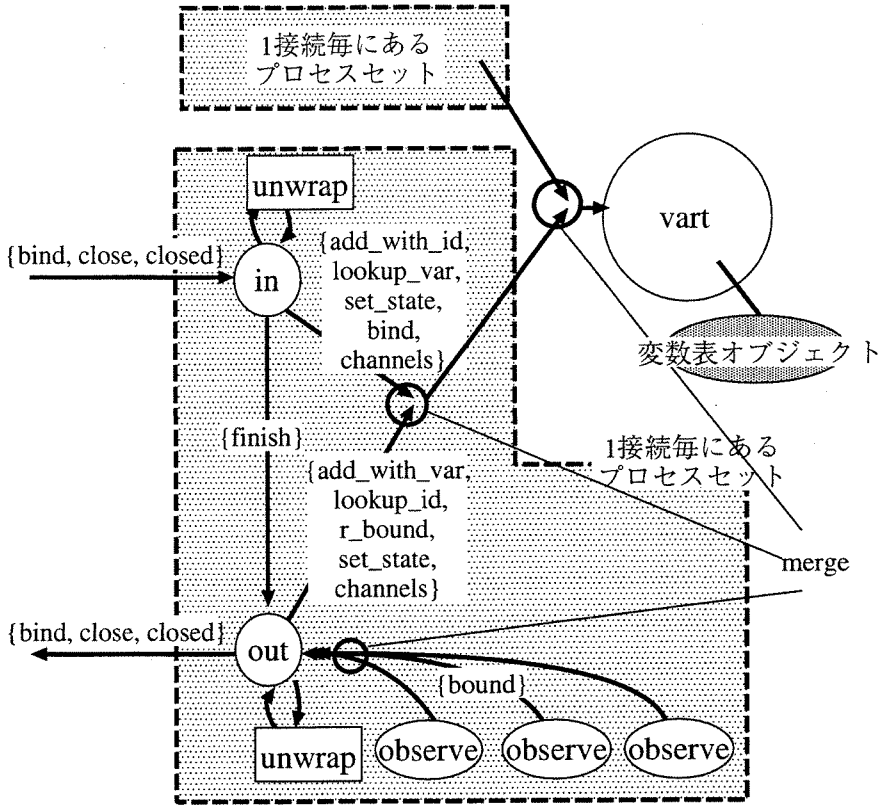



図 5.17: 分散論理変数プロトコル処理

**finish(ID)** リモートノード側で分散論理変数の総数がゼロであることを意味している. これを受け取ると, vart プロセスに対して channels メッセージを発行し, こちら側でも分散論理変数の総数を調べる. この総数がゼロならば, コネクションが終了したことを意味する. ネーミングプロセスとのコネクションでなければ (localhost とのコネクションならば), この接続を切断する.

**bind(ID,Term)** ID で示す分散論理変数の具体化メッセージ. これを受け取ったら, 手続き unwrap/4 を呼び出す. 第一引数には unwrap したい項, 第二引数は unwrap 後の項が返り, 第三引数には項の中に含まれていた分散論理変数 ID のリストで返る. 第四引数は unwrap が完了した事をしめす.

ここで得た ID リストは, add_with_ids/4 手続きに渡され, ID に該当する KLIC 変数を取得する. この時 vart プロセスには lookup_var メッセージが発行され, 該当変数がなければ add_with_id メッセージで新規に ID を登録して KLIC 変数を得る.

unwrap した項の ID はここで得た KLIC 変数に置換され, vart プロセスに対して bind(ID,UnwrapedTerm) メッセージと共に送られる. この bind メッセージが vart プロセスで処理されて初めてこのノードの KLIC 変数に具体化が反映される.

**out プロセス**

このプロセスは, ソケットへの項の書き込みを行なうプロセスである. このプロセスは in プロセスから finish を, observe プロセスから bound を受け取る.

**finish** このメッセージは, そのままリモートノードに送信する.

**bound(ID,Term)** ID が示す分散論理変数の具体化報告メッセージ. この具体化は, in プロセスが vart に送った bind メッセージの結果か, もしくはアプリケーションが具体化したかのいずれかである. まず, どちらなのかを vart プロセスに対して r_bound(ID,R) によって調べる. R が true であったなら前者のパターンであり, false なら後者である.

前者の場合は, リモートノードで具体化されたのを自ノードで反映しただけなので, プロトコルに従って, vart プロセスに対して set_state(closed) か set_state(closing) のいずれかを発行する.

後者の場合は, リモートノードに bind(ID,wrapedTerm) メッセージにより, 具体化を報告する. まず, Term を wrap/4 で wrap する. この時の wrap は, KLIC が提供する wrap とほぼ同じである. ただし, 変数が含まれていた場合, その変数のリストが第三引数に返る. このリストは手続き add_with_vars に渡され, 対応する ID に変換される. この時 vart プロセスには lookup_id メッセージが発行され, ID を得られなければ, add_with_var メッセージによって新規に分散論理変数が生成され ID が得られる.

**vart プロセス**

このプロセスは変数表オブジェクトを保持する唯一のプロセスである. メッセージを受け取ると, そのメッセージに対応するオブジェクトのメソッドを呼ぶだけである.

**observe プロセス**

このプロセスは分散論理変数を中断原因変数としてフックしていて, 具体化によって起き, out プロセスに対して, bound(ID,Term) メッセージを発行する. この Term はフックしていた変数が具体化した項である.

### 5.6.3 変数表オブジェクト

変数表オブジェクトは以下の機能を満たす Data オブジェクトである. ここでは API とデータ構造の説明を行なう. KLIC のジェネリックオブジェクト実装に関するドキュメントが不足しているので, 5.13 節に実装方法に関して補足している.

1. 分散論理変数へのユーザ参照チェックを行なう関数を定義する.
2. その関数を KLIC の GC の新たに追加実装した before_pcheck というタイミングで呼ばれるように登録する.

3. 分散論理変数を具体化したり, 状態を保持する.

4. ID 空間を管理する.

まず, API とデータ構造について説明しながら 3, 4 について説明する. その後に, 中継排除が起きるような例に沿って残りのデータ構造と 1, 2 を満たす C 言語の関数の説明を行なう.

### APIs

図 5.18 は, 以下のような一連のメソッド呼び出し (一通りとは限らない) の時のオブジェクトと KLIC ヒープ領域のデータ構造図である. これを参考にしながら, 以降の API の説明を読んでもらいたい.

1. `new(NewObj,Report)`
   変数表オブジェクトを生成する.

2. `new_connection(NodeA,Server,NewObj)`
   Subtable 構造体が生成される.

3. `lookup_var(@(0,nodeA),X,Result0,NewObj)`
   オブジェクト内に変化はない.

4. `add_with_id(@(0,nodeA),X,NewObj)`
   チャネル X が Channel 構造体のメンバから指される. 状態を示すメンバが OPEN になる.

5. `lookup_id(Y,nodeA,ID,Result,NewObj)`
   オブジェクト内に変化はない

6. `add_with_var(Y,nodeA,ID,NewObj)`
   チャネル Y が Channel 構造体のメンバから指される状態を示すメンバが OPEN になる.

API の呼び出しは, これから示す API の引数の他に最初にオブジェクト自身を与えなければならない. 例えば, `channels/2` とあったら, `generic:channel(Obj,CID,NewObj)` と呼び出さなくてはならない. オブジェクト自身を省略するのは KLIC マニュアルの記述法にならっているためである.

以下は API の一覧である. API の説明文の中で, 引数に NewObj と出てきた場合はメソッドの作業後のオブジェクトが返る変数であり, 説明を省略している.

- `new/2`
- `new_connection/3`
- `lookup_id/5`
- `add_with_var/4`

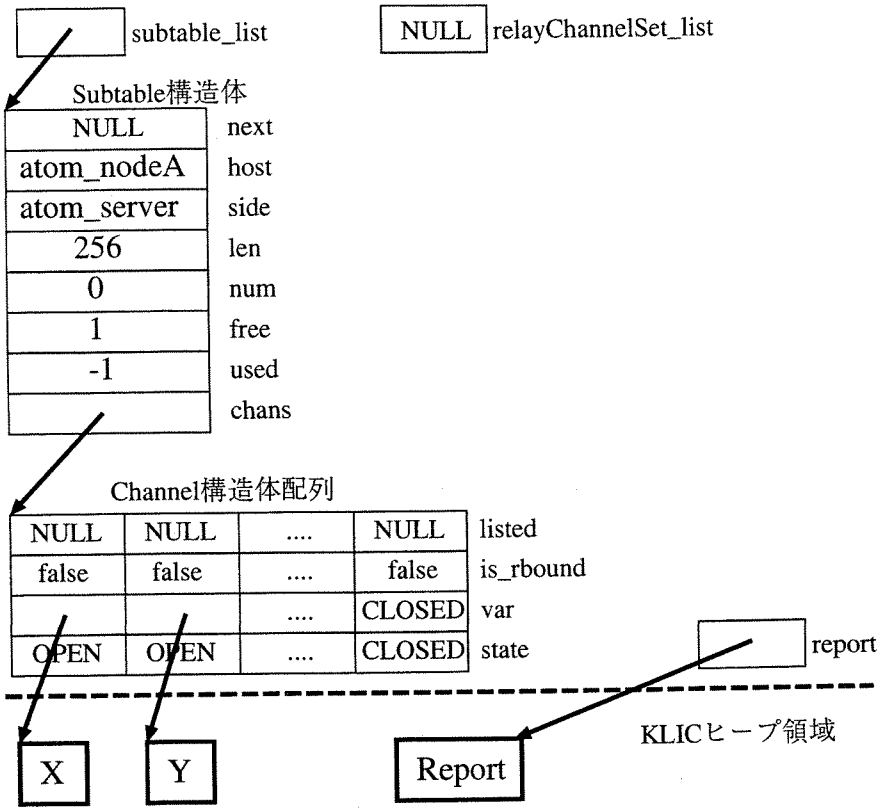



図 5.18: 変数表オブジェクトのデータ構造 1

- `lookup_var/4`
- `add_with_id/3`
- `is_rbound/2`
- `bind/4`
- `set_state/3`
- `get_state/2`
- `channels/2`
- `equal/3`
- `close_connection/2`

**new/2** −NewObj −Report

変数表オブジェクトを生成する. このオブジェクトは 1 ノードに 1 つでなければならない (2 個以上の生成は抑制していない). Report は中継排除プロトコルに必要な分散論理変数の ID が 2 個以上のセットが具体化されてくる遅

延具体化ストリームであり終端はない. 書式を以下に示す.

$$\text{Report} = [\ [@(\text{IntID}, \text{ConnectionID}) | \text{IDs}]\ |\ \text{Report2}]$$

また, `before_pcheck` タイミングで呼ばせる関数を `before_pcheck` 関数ポインタ配列に代入する.

**new_connection/3** +ConnectionID +Side −NewObj

コネクション ID を受け取って, Subtable 構造体を作る. Side は, このオブジェクトが存在するノードがコネクション開設側 (`client`) か被開設側 (`server`) かを示す. Subtable 構造体は以下のようなメンバを持つ.

**next** Subtable 構造体リストの次の Subtable 構造体

**host** コネクション ID を保持する

**side** Side を保持する

**len** このテーブルが管理する ID 空間の長さ*2 (`chans` の先にある Channel 構造体の長さ). ID 空間は, `side` が `client` ならゼロから始まる偶数. `server` なら 1 から始まる奇数. `chans` では両方の ID 空間に関連付けられた分散論理変数が入るので, 長さは 2 倍.

**free** 管理する ID 空間の要素 (整数) のうち, 未使用のものの最小値.

**used** 両方の ID 空間の要素のうり, 使用中のものの最大値.

**chans** 両方の ID 空間に関連付けられた分散論理変数に対応する Channel 構造体配列へのポインタ.

**lookup_id/5** +Chan +ConnectionID −ID −Result −NewObj

Chan を受け取り, そのチャネルに該当する ID を返す. 該当するとは, 管理する 全ての Channel 構造体のうち, Chan と同じ KLIC 変数を `var` メンバに持つ構造体を指す. 返す ID は, その Channel 構造体が属する Subtable の `host` メンバの値と, その Channel 構造体の配列におけるインデックスの組のことで,

$$@(host, index)$$

で表す. このような Channel 構造体が見つからない場合は, Result に `false` を返す (当然見つかった場合は `true`). `false` である場合 ID は未定義である. Channel 構造体は以下のようなメンバを持つ.

**listed** RelayChannelSet 構造体へのポインタ.

**is_rbound** この Channel 構造体の var メンバが bind メソッドにより具体化されたかが true か false で保持されている. true の場合は該当する分散論理変数がリモートノードで具体化されたことを意味する.

**var** この Channel 構造体が表す分散論理変数に該当する KLIC 変数へのポインタ.

**state** この Channel 構造体が表す分散論理変数の状態を保持するメンバ. このメンバが CLOSED の場合は, var と is_rbound メンバは無効な値である. 特に var メンバに格納されたポインタを手繰るとセグメンテーションフォルトになる可能性が高い.

**add_with_var/4** +Chan +ConnectionID −ID −NewObj

一般に lookup_id が失敗した場合に, 新たに分散論理変数を割り当てるために呼ばれる. host メンバが ConnectionID であるような Subtable 構造体の管理する ID 空間の空き ID を割り当てる. これは host と free メンバの組である. また, Channel 構造体配列の free 番目の Channel 構造体の var メンバに Chan へのポインタを代入し, state メンバを OPEN にする. 最後に, used と free メンバを正しい値に書き換え, num メンバをインクリメントする.

**lookup_var/4** +ID −Chan −Result −NewObj

ID に該当する Channel 構造体の var メンバが指す KLIC 変数を Chan に返し, Result には true を返す. そのような Channel 構造体がないか, state メンバが CLOSED になっていたら, Result に false を返す. false の場合は Chan は普通の KLIC 変数のままである.

**add_with_id/3** +ID −Chan −NewObj

ほとんどの場合 lookup_var で失敗した場合に, ID で表す分散論理変数に該当する KLIC 変数を作りたい時に呼ぶ. ID に該当する Channel 構造体の var メンバを Chan に返し, state メンバを OPEN にする. さらに, Subtable 構造体の num メンバをインクリメントする.

**is_rbound/2** +ID −Result

ID に該当する Channel 構造体の is_rbound メンバが Result に返る.

**bind/4** +ID +Chan +Term −NewObj

このオブジェクトが管理する全ての var メンバの指す KLIC 変数のうちで, Chan と同じ KLIC 変数を Term で具体化する. この時, ID に該当する Channel 構造体の is_rbound メンバを true にする.

**set_state/3** +ID +State −NewObj

ID に該当する Channel 構造体の state メンバに State を代入する. State はアトムでなければならない.
また, closed だけが予約されていて set_state(ID,closed) とすると, その ID で表される分散論理の廃棄を意味し, Subtable 構造体の num メンバをデクリメントし, Channel 構造体の listed メンバの指す先を free する.

**get_state/2** +ID −State

ID に該当する Channel 構造体の state メンバを State に返す.

**channels/2** +ConnectionID −Num

ConnectionID に該当する Subtable 構造体の num メンバを Num に返す.

**equal/3** +Chan1 +Chan2 −Result

Chan1 と Chan2 が同じ KLIC 変数であるかを調べ, 結果を true, false で Result に返す. C 言語レベルのポインタ値で比較する.

**close_connection/2** +ConnectionID −NewObj

ConnectionID に該当する Subtable 構造体の Channel 構造体配列の全ての要素を未使用状態にする. ただし, 使用状態だった Channel 構造体の var メンバの指す KLIC 変数は空リストで具体化する. この時書き込み衝突が起きるかもしれないが, それは上位層の設計ミスによる書き込み衝突なので無視して書き込む.

### 中継排除に関する関数とデータ構造

図 5.18 で X と Y が単一化が起き, KLIC の GC が起きて, before_pcheck タイミングの直前の状態が図 5.19 である. 1 番目と 2 番目の Channel 構造体は同じ KLIC 変数を指していることが分かる. ここで, 中継排除対象の分散論理変数を検出する関数 (detect_report_copy) が呼ばれる. 以降は関数の動きをプログラムと共に解説していく.

関数名のとおり, 大きく detect フェーズ, report フェーズ, copy フェーズの 3 フェーズに分けられる.

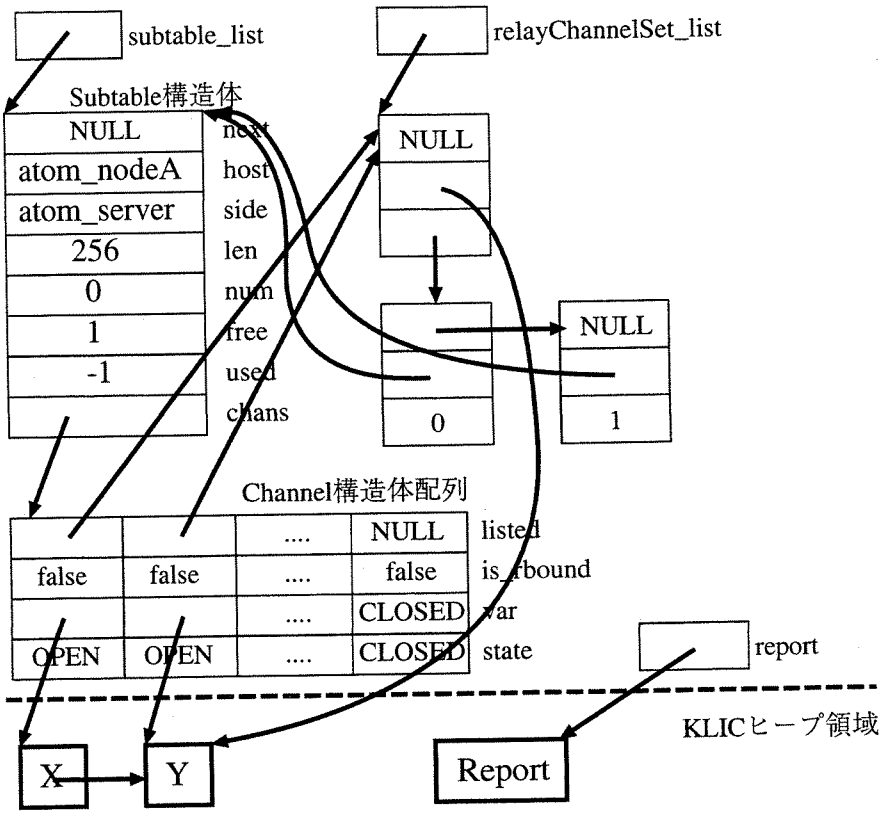



図 5.19: 変数表オブジェクトのデータ構造 2

**detect フェーズ**

まず, 全ての Subtable 構造体の Channel 構造体配列を調べ, 同じ KLIC 変数を指している Channel 構造体を探すフェーズである. 見つかった場合は, RelayChannelSet 構造体を作り, 検出した変数に対応する RelayChannel 構造体を次々につなげていく. この RelayChannelSet 構造体もリストをなしており, 次の report フェーズで処理する. この時点で先ほどの図 5.18 は, 図 5.19 のようになっている. 図では 2 変数が単一化しているが, これが 3 変数以上なら RelayChannel 構造体は 3 個繋がることになる.

また, この detect フェーズでもうひとつ重要な点は, KLIC 変数を比較する時に正規化する `VT_deref` 関数 (図 5.21 を参照せよ) である. まずは, KLIC データのヒープ上での様子を示す. 変数や主なデータは図 5.20 のようになっている. 正規化すると `VT_deref` の第二引数は太線で書いたセルを指すことになる. これは未定義変数同士の単一化が起きたときにポインタ鎖の一番最後を見ることで両変数が等しいかが判断できるためである. リストや, 各データの bit 表現などについては Inside KLIC [11] を参照せよ.

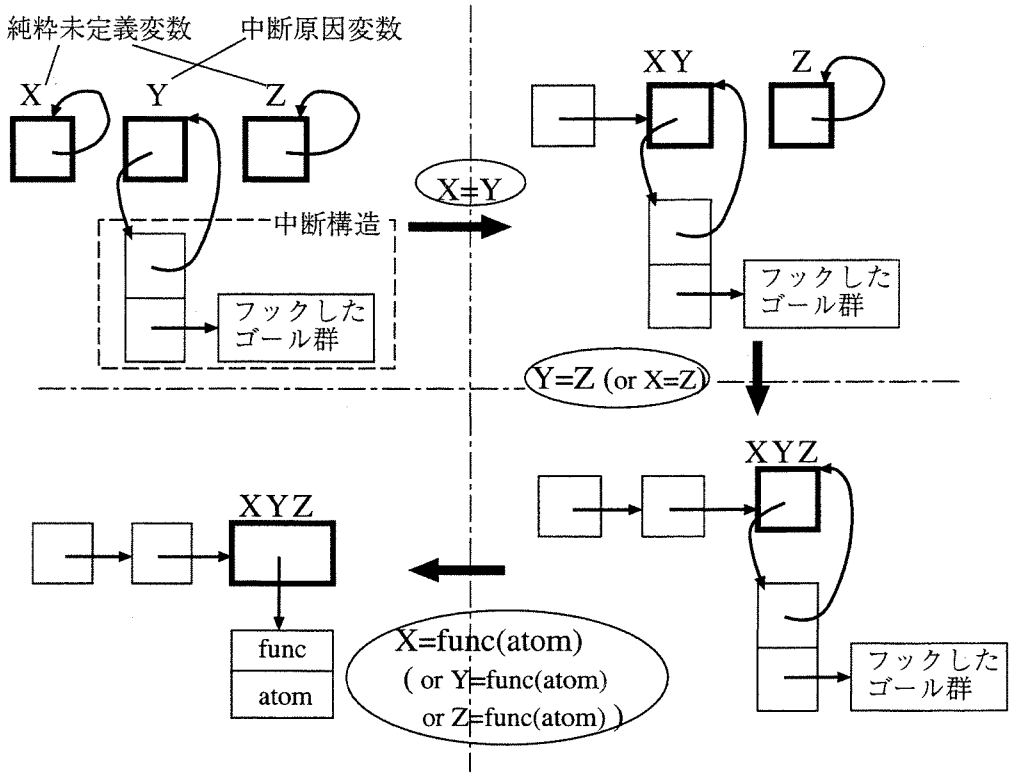



図 5.20: KLIC ヒープの様子 1

**report フェーズ**

detect フェーズで作成した RelayChannelSet 構造体リストに挙げられた変数に KLIC 変数ユーザ参照の有無をチェックする. ある RelayChannelSet 構造体の `var` メンバについてユーザ参照がなかったなら, その構造体に連なる RelayChannel に書かれた情報を元に ID を作成し, KLIC 変数 Report に ID のセットを具体化する. 具体化したら RelayChannelSet 構造体などは free し, Channel 構造体の `listed` メンバを REPORTED とする. 次回の `detect_report_copy` 関数呼出時には報告しないようにするためである. もしユーザからの参照があれば, 何もせずに次回呼出時に再びチェックする. API の `set_state` でも述べたが, 対応する Channel 構造体の state が close になったら, これらのリストも free する.

図 5.22 は, 繰返し検討してきたユーザ参照をチェックする関数である. 参照されていない時だけゼロを返す. 新領域にコピーされていると 2 を返し, `observe`(具体化監視ゴール) 以外がフックしていると 1 を返す. ここに出てくる `susprec` 構造体なるデータ型のポインタは, 図 5.20 で出てきた中断構造へのポインタである. do-while 文内ではフックしたゴール群に `observe` ゴール以外のゴールが無いかチェックしている.

```
static int VT_deref(q var, q *derefered)
{
  q term = var;
  if( ! isref(term)){
    *derefered = term;
    return 1;
  }
  for(;;){
    q tmp = derefone(term);
    if(isref(tmp)){
      if(term==tmp){
        *derefered = term;
        return 0;
      }else if(term==derefone(tmp)){
        *derefered = term;
        return 0;
      }else term = tmp;
    }else{
      *derefered = term;
      return 1;
    }
  }
}
```

図 5.21: KL1 項の正規化関数

```
static int VT_refcheck(q var)
{
  declare_globals;
  if(VT_WITHIN_NEW_SPACE(var)) return 1;
  if(isref(var)){
    q tmp = derefone(var);
    if(isref(tmp) && var == derefone(tmp)){
      struct susprec *s = suspp(tmp);
       if( ! is_generator_susp(s->u)){
        struct hook *second_hook = s->u.first_hook.next;
        struct hook *h = second_hook;
        struct hook dummy;
        struct hook *last = &dummy;
        do{
          if(h->u.g->pred != &predicate_dklic4_xobserve_3)
                                                  return 2;
          h = h->next;
        }while( h != second_hook);
      }
    }
  }
  return 0;
}
```

図 5.22: ユーザ参照チェック関数

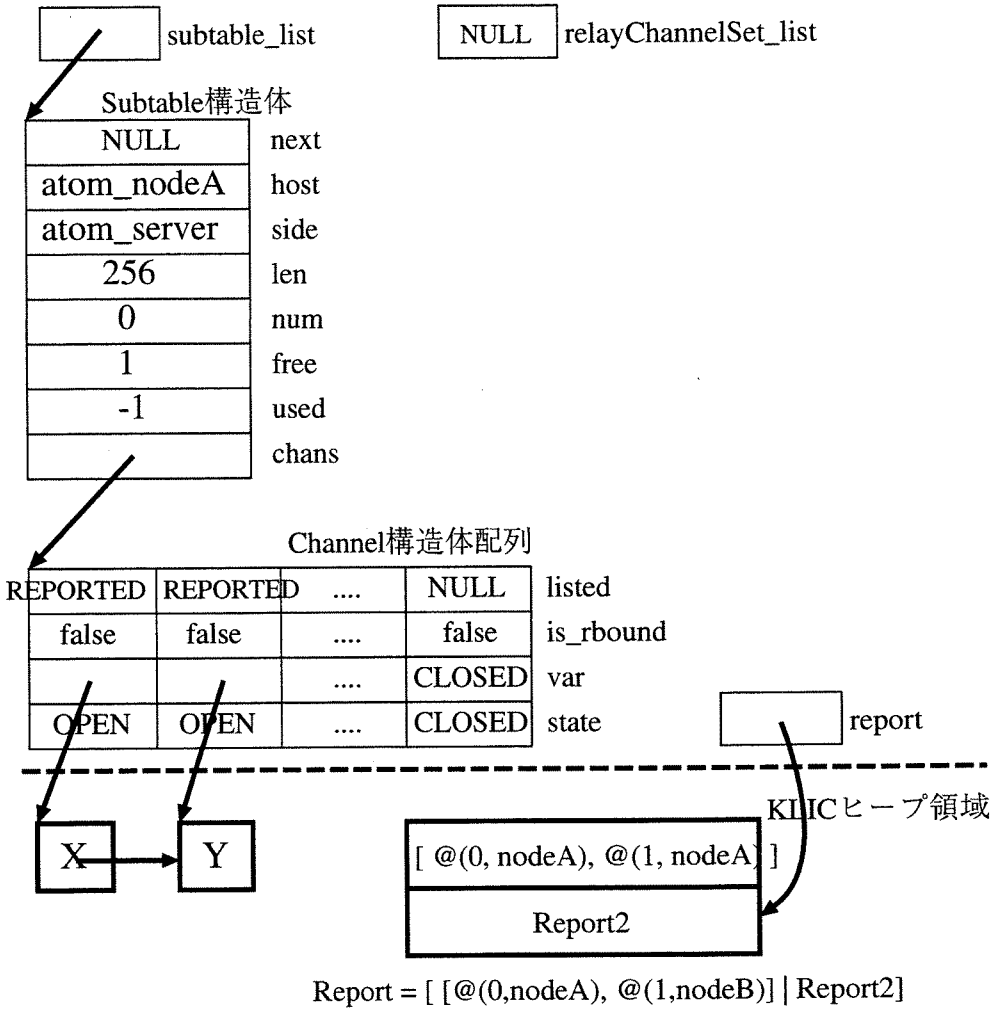



図 5.23: 変数表オブジェクトのデータ構造 3

**copy フェーズ**

このフェーズでは, 分散論理変数にあたる変数と一緒に, それにフックしているゴール群をとコピーする. KLIC が提供している `copy_one_term` なる関数を呼んで変数を与えるだけである. ただし, 目的の変数へのポインタを格納した変数のアドレスを渡さなくてはならない.

## 5.7 ネーミング層の仕様と設計

ネーミング層では, 分散論理変数への名前付けを行なう. ここではチャネルの登録という呼び方をする. 名前付けが, 特定のサービスを提供するサーバへのチャネルの登録と見なせるからである. 同様に名前ではなく, サーバ仕様と読み換えることにする. また, サーバ仕様によるチャネル lookup の解決も行なうが, 複数ノードのネーミングプロセスが連係して解決することも可能である. この連係は, P2P に基づいたアドホックな DKLIC ネットワークを形成する役割も果たす.

本節では, DKLIC ネットワークの形成, チャネルの登録, チャネルの lookup, lookup の複数ネーミングプロセスによる解決について述べる.

### 5.7.1 ネットワーク形成

ネーミングプロセスは各ノードに一つ必ず動いているが, これらのネーミングプロセスは, 起動時に IP マルチキャストによって, 自分の存在を宣伝する. マルチキャストを受け取った起動済みのネーミングプロセス (ネーミング A) は, マルチキャストの送信元 (ネーミング B) に対し, TCP コネクションの開設を要求する. 開設後は, ネーミング A は接続先リストにネーミング B を加え, さらにネーミング B に対してネーミング A (自分) を接続先リストに登録するように要請する. この操作によって互いに相手を認識する. 図 5.24 を参照せよ.

### 5.7.2 チャネル登録

チャネルの登録は以下の API を用いる.

`register(Spec, Channel)`

ここで Spec とは登録しようとするサーバの仕様を意味する. 形式は, 以下のような属性名をファンクタ名とし, 引数をその属性名の値とするようなファンクタのリストである. ファンクタ名は重複してはならないし, 引数は 1 個でなくてはならない. ここに記述した属性以外の属性 *attribute* は, *attribute*(*) を指定したと見なす. *とは全ての値にマッチする記号である.

`[name(http_daemon/3), http_version(1_0),...]`

登録に成功すると Channel は `normal(S)` に具体化される. この S には `listen` メッセージを具体化していく. 図 5.3 や図 5.4 を参照せよ). 次節で述べるがクライアントの `lookup`

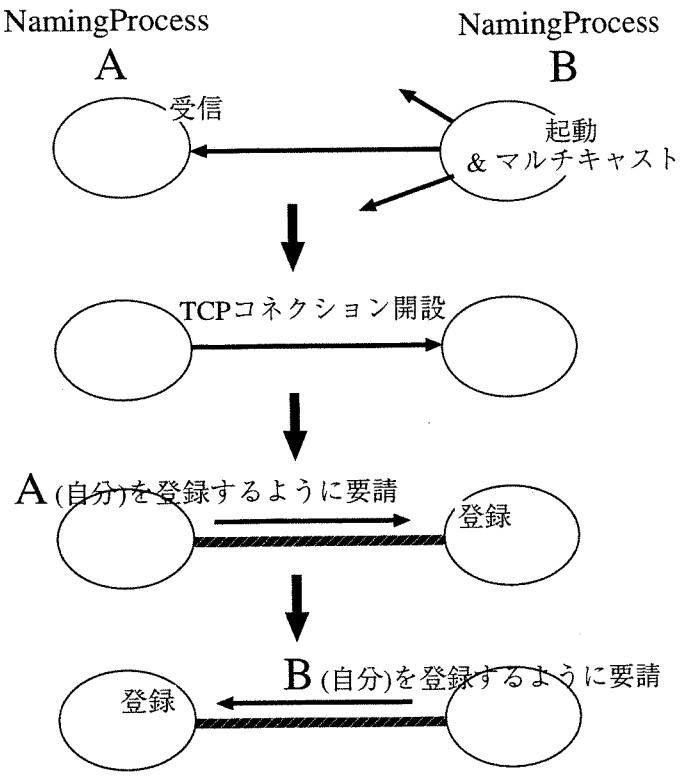



図 5.24: ネーミングプロセスの起動

要求はサーバ側のネーミングプロセスに貯められる. サーバが listen メッセージを具体化すると, その引数がクライアントに渡される.

また, API は内部で以下のように自ノードのネーミングプロセスに register メッセージを送っている. ネーミングプロセスは, このリストを保持する.

```
register(Spec, Channel) :-
  dklicio(H,[connect(localhost,Port,
              normal([register([channel(Channel)|Spec],_)]))]).
```

### チャネルの delete

Jini や DNS などは, 登録済みエントリが不要になった場合は明示的に削除する必要があるが, DKLIC におけるネーミングでは登録したチャネルを単に閉じるだけでよい. 登録したチャネルが閉じるとネーミングがそれを検出し, エントリを削除する. これはサーバがサービスの提供を止めることが, チャネルが閉じてゴール群 (DKLIC ネットワーク) から切り離されるという KL1 のセマンティクスと同義であることを意味する.

### 5.7.3 チャネルの lookup

チャネルを lookup するには以下の API を用いる.

lookup(Spec,Result)

Spec で示す仕様に合致するサーバが見つかると, そのサーバへのチャネルが normal(Channel) で返る. 見つからないと abnormal が返る. Spec は前節の register の引数と同じ形式である.

lookup は内部で以下のように自ノードのネーミングプロセスに lookup メッセージを送っている. Address は lookup を発行したクライアントノードのアドレスである. lookup を複数ノードで連係して解決する時に発行元のアドレスが必要となるためである. Hop は整数で, 複数ノードのネーミングプロセスが連係してこの lookup を解決するときに, lookup メッセージを転送していく. この転送回数の上限を定める. 現在は, ユーザは設定ファイルから設定する仕様になっている.

```
lookup(Spec,Result) :-
  dklicio(H,[connect(localhost,Port,
              normal([lookup(Address,Hop,Spec,Result)]))]).
```

### 5.7.4 複数ノードのネーミングプロセスによる連係

これは lookup を受けたプロセスが, 自分の管理内では見つからない場合には, 周辺の TCP コネクションを開設しているネーミングプロセス全てに lookup を転送する. ことで実現している. この時複数のノードで lookup が成功した場合に, どのチャネルを用いるか選択しなくてはならない.

図 5.3 や図 5.4 を見ると discovery や adopted といったメッセージがあるが, discovery はネーミングプロセスによって具体化されるメッセージで lookup が成功したことを意味する. adopted は lookup のクライアント (ネーミングプロセスではない) がその lookup の結果を最終的に採用したことを通知するメッセージである. ネーミングプロセスは, クライアント側から adopted とサーバ側から listen が揃った時に各メッセージの引数のチャネル同士を連結 (単一化) する.

この一連のやりとりを図に示す. 図 5.25 はこのネットワークの接続状況である. ここで登録されたサーバは, クライアントの lookup した Spec に一致するものとする. 図 5.26 はノード A のネーミングプロセスが, 見つけられずに lookup を転送した様子である. ノード B も見つけられないが, ホップ上限数が 0 になっているので, Channel2 には abnormal を返す. 一方ノード C ではサーバを発見できたので, discovered メッセージを返す.

図 5.27 は, クライアント, ノード A のネーミングプロセス, ノード C のネーミングプロセス, およびサーバが交わすメッセージの様子である. 図 5.27 は, ノード B でも lookup が成功した場合である. 特徴的な点は, 採用しなかった方をキャッシュしている点である. 次回ノード A のネーミングプロセスに lookup があった場合は, このキャッシュが使う. この場合は, discovered が送られた後からの一連のプロトコルが始まる. また, 使ったキャッシュは消す.

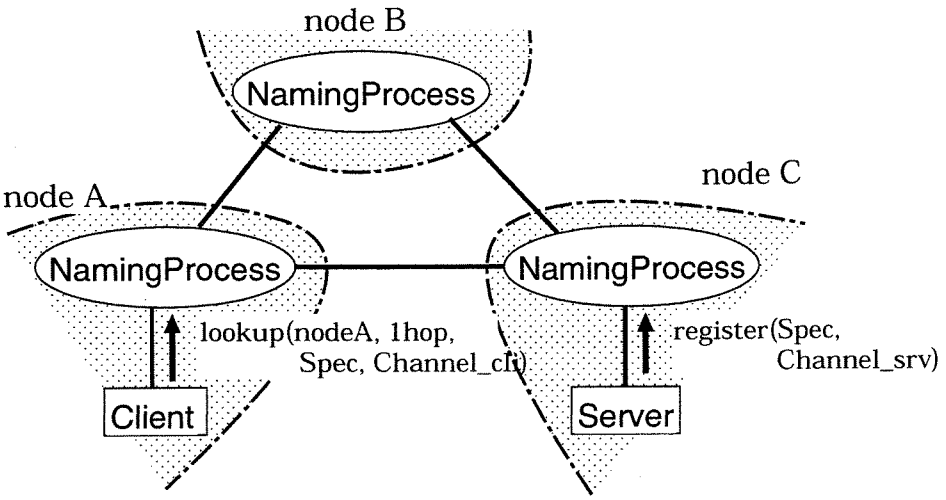



図 5.25: lookup の解決と転送 1

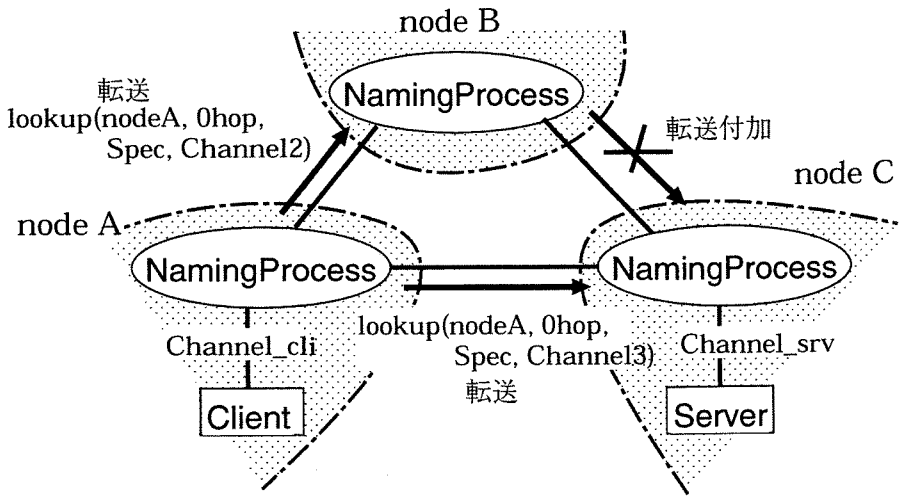



図 5.26: lookup の解決と転送 2

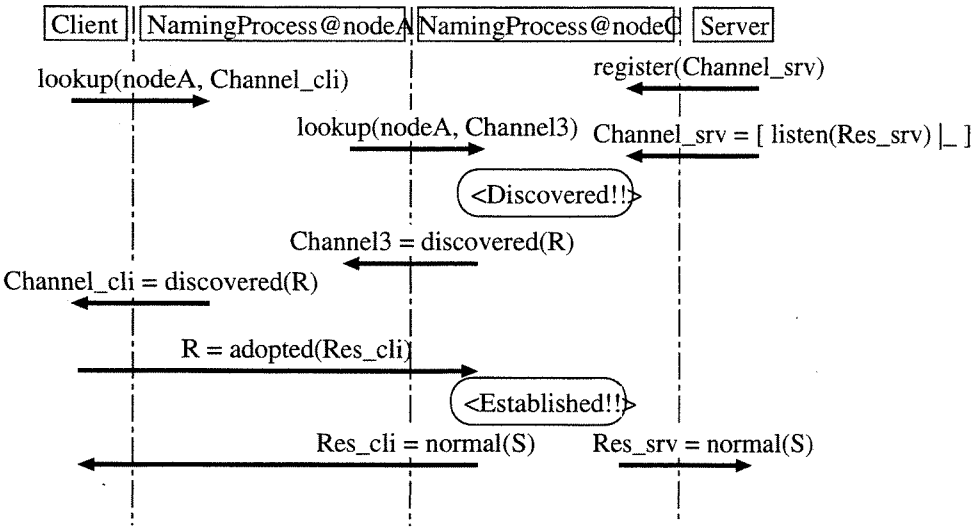



図 5.27: lookup 成功後の接続確立

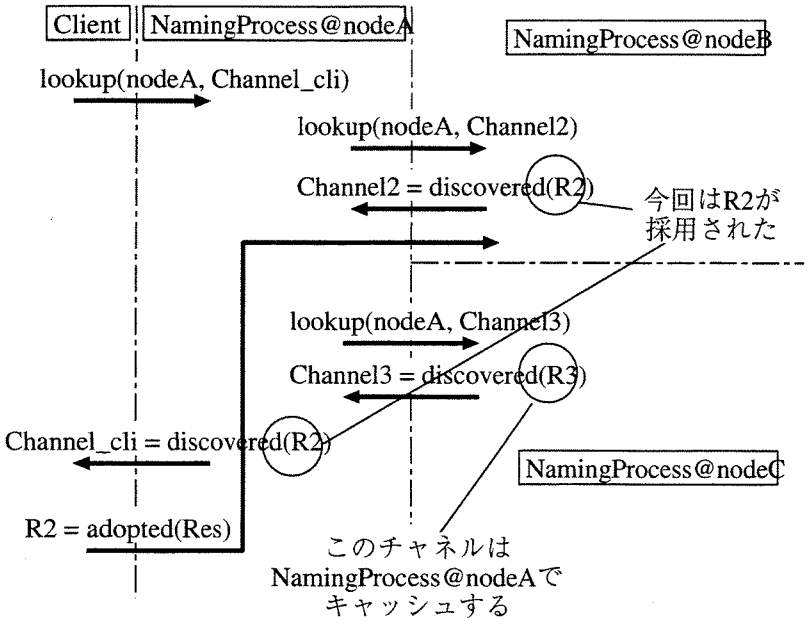



図 5.28: 接続確立とキャッシュ

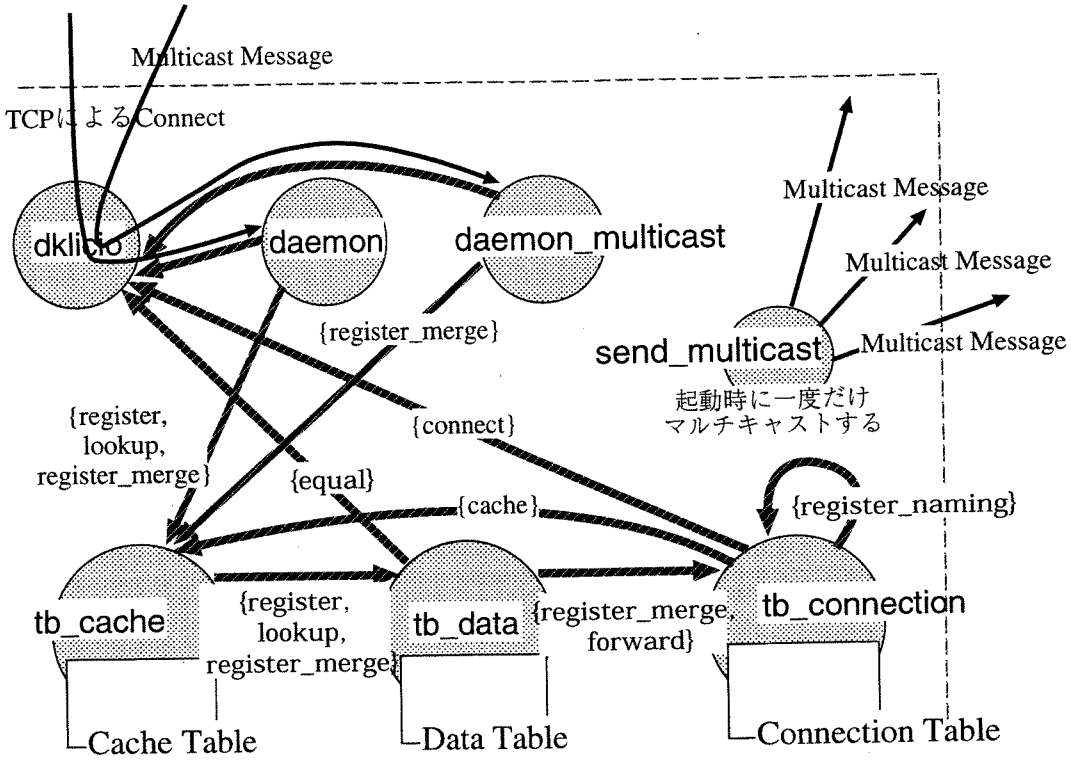



図 5.29: ネーミングプロセスの構造図

## 5.8 ネーミング層の設計と実装

### 5.8.1 ネーミングプロセスの設計

ネーミング層の実装であるネーミングプロセスは, 図 5.29 のように設計した. まず各プロセスの役割を説明し, 以降はネーミングプロセス起動時の動き, チャネル登録やチャネル取得時の動きについて述べていく.

#### dklicio プロセス

分散論理変数層の実装である dklic モジュールへのストリームを管理するプロセス. 現在は単にマージし, 中継しているだけである. 全てのネットワークへの通信はこの dklic モジュールを通して行なっている. `discovery` もここを通しているが図が複雑になるので省略している.

#### daemon_multicast

マルチキャストによるメッセージを受け取り, その送信元アドレスを元に他ノードとの TCP コネクション確立を行なう.

マルチキャストの実装

従来の KLIC にはマルチキャストに関する機能がなかったので追加した. API は以下の 2 つである.

**send_mcast** +inet(G,P) −Res
G は参加する IP マルチキャストグループを示す IP アドレスである. この IP アドレスは, 224.0.0.1 ～ 239.255.255.255 の範囲内でなければならない. DKLIC では 239.137.194.111 を使う. また, P はポート番号であり 8181 を利用することにした.

`send_mcast` では connect を用いることとした. こうすることでこのソケットに繰返しメッセージを流すことができるため, KL1 プログラムとの相性が良い. Res にはこの connect が成功した場合 `normal(S)` が返り, 失敗の時は `abnormal` が返る. S には `put/1` メッセージを流すことができる. `put` の引数には整数だけを与えることができる.

**bind_mcast** +inet(G,P) −Res
`send_mcast` で述べたように, IP アドレス 239.137.194.111, ポート番号 8181 にバインドする. `Res` にはバインドが成功すると `normal(S)` が返り, 失敗すると `abnormal` が返る. S には `get/2` メッセージを流すことができる. 第一引数は, `put` メッセージが流した整数が具体化し, 第二引数にはそのパケットを送信した送信元の IP アドレスが具体化する.

**daemon**

TCP によるコネクションを受けるプロセス. 複数のプロセス (ネーミングプロセス, サーバプロセス, クライアントプロセス) からのチャネル登録, チャネル取得などのメッセージ (5.7.2 節や 5.7.3 節を参照せよ) をマージし, `tb_cache` プロセスに渡す.

**tb_cache**

サーバ仕様と 5.7.4 節で述べたキャッシュチャネルの対応表を保持するプロセス. `register`, `lookup`, `register_merge`, `cache` メッセージを受け取る. `register`, `register_merge` メッセージは何もせずに `tb_data` プロセスに渡す.

**cache(Spec,Channel)** Spec に示されたサーバ仕様と Channel を表に登録する (5.7.4 節を参照せよ). 登録する時に, Spec と Channel の ID (5.6.1 節を参照せよ) の全てが同一のエントリが既に表中に存在したら, 登録しない.

**lookup(Spec,Res)** Spec は前節で述べたようなサーバ仕様である. `tb_cache` は Spec がマッチするエントリが保持している表内に存在すれば Res にそのチャネルを返し, エントリを削除する. このチャネルは 5.7.4 節で述べたように `discovered` が送られた後のプロトコルから始める.

### tb_data

サーバプロセスが登録したチャネルとサーバ仕様の対応表を保持するプロセス. register_merge, register, lookup メッセージを受け取る. register_merge は何もせずに tb_connection に渡す.

**register(Spec,Res)** Spec に示されたサーバ仕様を保持する. register メッセージの Spec は必ず1つ channel 属性を含まなくてはならない. Res には registered (登録完了) か登録時に, Spec と Channel の ID の全てが同一のエントリが既に表中に存在したら, 登録せずに, already_registered (既に登録済) を返す.

**lookup(Spec,Res)** サーバプロセスが登録したチャネルに対してヒットする. ヒット後は listen メッセージが流れて来るまで待って, listen メッセージ中のチャネルを Res に返す. また, ヒットしなかった場合は lookup メッセージを forward メッセージに改名して tb_connection に渡す. これは tb_connection が保持している全ての他ノードのネーミングプロセスに lookup を転送することを意味する. tb_connection は複数ヒットしても1本のチャネルしか返してこない.

### tb_connection

他ノードのネーミングプロセスに繋がるチャネルを保持するプロセス. forward, register_merge, register_naming を受け取る. register_naming は自らが発行するメッセージである.

**forward** tb_data からのチャネル取得メッセージ (lookup) の転送要求である. 保持する他ノードとのチャネルを全て用いて lookup メッセージを転送する. この結果, 複数のチャネルを取得できた場合は, 1つだけ選び, 他は tb_cache へキャッシュ要請メッセージ (cache) を発行する. 転送する時には lookup メッセージ中のホップ上限パラメータを減らして転送する. ただし, ホップ数が 0 のメッセージは転送しない.

### 他ノードとのメッセージ送受信

TCP による conecct は dklicio プロセスのみが行なえるが, 各プロセスはそのコネクション上に生成されるチャネル (分散論理変数) によって特定のプロセスを介さずに自由に他ノードのプロセスと通信する点に注意しなくてはならない. (当然だが, TCP コネクションがなければそのノードとの間にチャネルを持つことはできない)

これは, 図 5.30 のように, チャネルに対してチャネルを含む項で具体化した場合新たな通信路が暗黙の内に設定されることによる. この過程を全て図中に含めていると冗長なので省略する. 唐突に通信路が生まれている場合はこのような場合だと理解しなければならない.

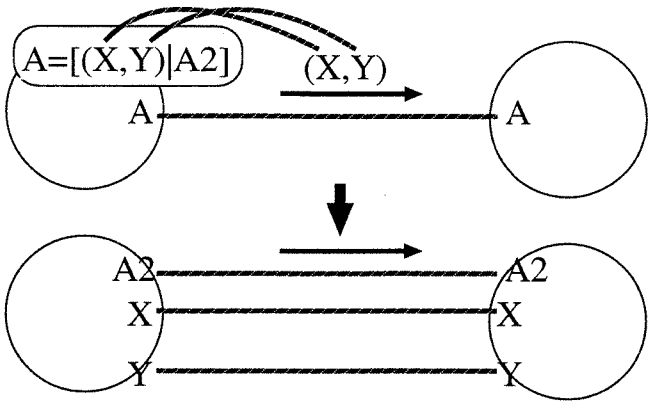



図 5.30: チャネル含むメッセージ

### 5.8.2 コネクション確立フェーズ

前節で述べたコネクションの確立は, ネーミングプロセス上では図 5.31 のように設計する. 受信側 (ノード A) の `daemon_multicast` プロセスはこのマルチキャストの送信元 (ノード B) に対して dklicio コネクション (TCP コネクション) を開設し, ノード A の Connection Table に対して `register_naming` を送り, ノード B の Connection Table に対しては `register_merge` メッセージを送り, ノード AB 間のコネクションを相互に登録する.

`register_merge` メッセージの引数にはノード A (自分) のアドレスと, チャネル X とチャネル Y の組を渡す. この Y チャネルはノード A の Connection Table に送る `register_naming` メッセージの引数に渡される.

### 5.8.3 チャネルの delete

前節で述べたチャネルの自動 delete は, 各メッセージ処理の時に空リストかどうかをチェックし, 空リストなら廃棄する lazy な GC として設計した. GC のために全体の処理が停止することを避けるためである.

## 5.9 実行例

### 5.9.1 動作に必要な環境

本実装は以下のようなファイル構成となった

- naming.kl1(ネーミングモジュール)
- dklicio.kl1 (dklicio の KL1 部分)
- vart.c vart.h (dklicio のジェネリックオブジェクト)

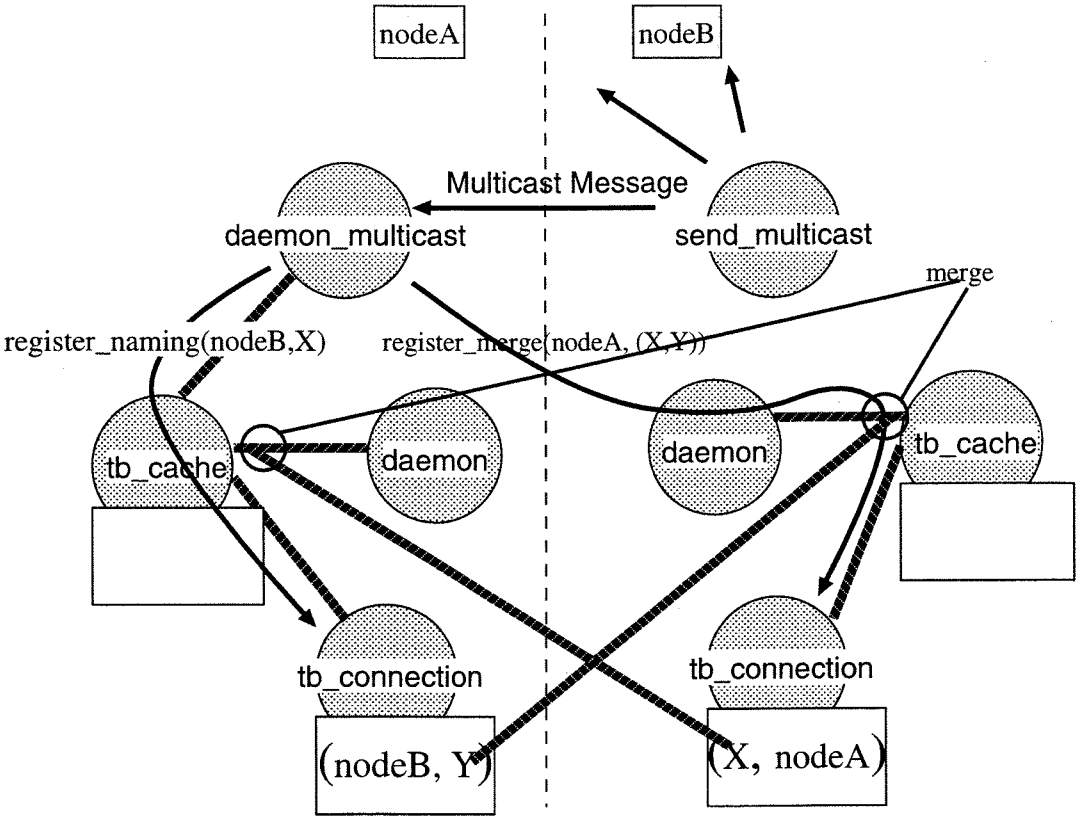



図 5.31: ネーミングプロセス間接続の確立

また, これらの他に klic-3.004-extio-shared 版の KLIC をインストールする必要がある. これは klic-3.003-extio-shared 版にバグフィクスや今回実装したマルチキャスト機能や GC における新規の関数フックなどを追加したバージョンである.

### 5.9.2 2 ノードによる DKLIC アプリケーションの実行例

ここでは, 2 ノードのネーミングプロセスが連係する様子を確認する. ここで用いるのは 5.12.1 節, 5.12.2 節, 5.12.3 節に示したプログラムである. 例えば intlist.kl1, sum.kl1 のようなサーバクライアントアプリケーションを図 5.32 のように稼働させたい場合は, rigotte と soleil の dklic.conf を各々以下のように書き込む.

nodeA 側の dklic.conf

```
port(8181).
host_address("soleil.xxx.yyy.zzz").
group_address("239.137.194.111").
```

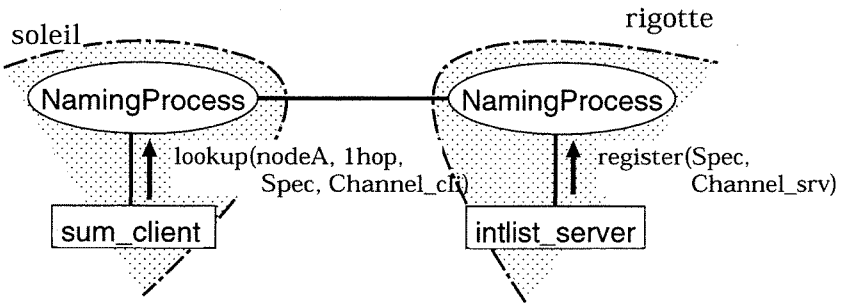



図 5.32: 2 ノードによる実行

nodeA 側の dklic.conf

```
port(8181).
host_address("rigotte.xxx.yyy.zzz").
group_address("239.137.194.111").
```

そして以下のようにネーミングプロセスをコンパイルする.

ネーミングプロセスのコンパイル

```
rigotte % klic -o ns ns.kl1 naming.kl1 dklicio.kl1 vart.c
soleil % klic -o ns ns.kl1 naming.kl1 dklicio.kl1 vart.c
```

さらに, アプリケーションプログラムをコンパイルする.

DKLIC アプリケーションのコンパイル

```
rigotte % klic -o srv intlist.kl1 naming.kl1 dklicio.kl1 vart.c
soleil % klic -o cli sum.kl1 naming.kl1 dklicio.kl1 vart.c
```

これらを実行するには, 各ノードでネーミングプロセスを立ち上げる必要がある. 図 5.33 は, ネーミングプロセスを右側 (マシン名 soleil) で起動した後に, 左側 (マシン名 rigotte) で起動した様子である. (`import/export` メッセージは, ソケット通信のダンプである)

ここで soleil で `intlist` サーバを探すクライアントを起動すると, 当然チャネルを取得できないが, `intlist` サーバを rigotte で起動した後では取得できる (図 5.34). これによってネーミングプロセスが連携していることが確かめられる. また, サーバから整数リストが徐々に具体化していることも分かる.

### 5.9.3 3 ノードによる DKLIC アプリケーションの実行例

ここでは, キャッシュが有効に作用していることを確認する. 図 5.35 が全体図である. 先ほどと同様にマシン brie の dklic.conf を設定する. 2 ノードでの実行から引き続いて,

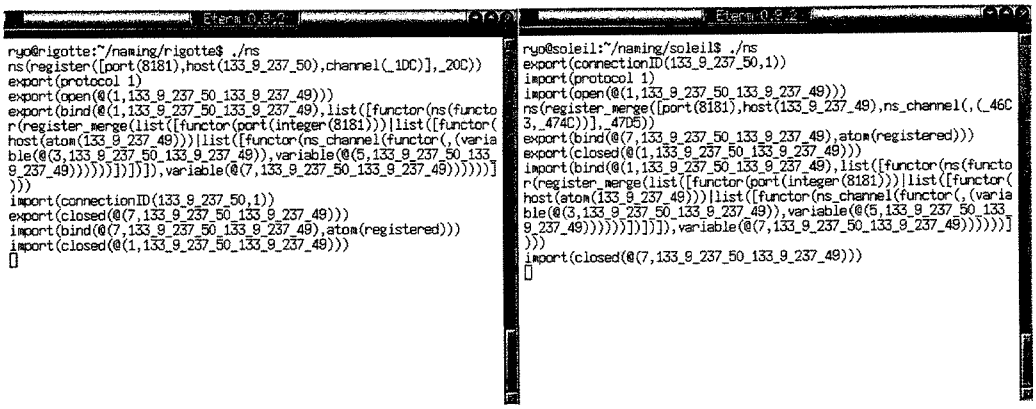



図 5.33: ネーミングプロセスの実行


図 5.34: ネーミングプロセスの連携

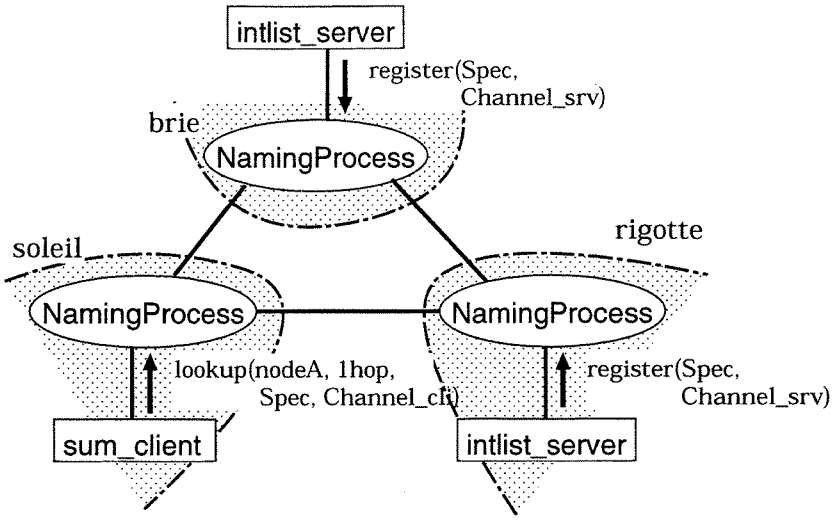



図 5.35: 3 ノードによる実行

brie でネーミングプロセスと `intlist` サーバを起動する．再び soleil でクライアントを実行すると 2 つのサーバが反応してする．もう一度クライアントを実行すると片方のサーバしか反応しない．これは前回のクライアント起動時に片方のサーバチャネルが soleil のネーミングプロセスでキャッシュされていたことを示している．

## 5.10 関連研究

類似研究としては，Mozart[12, 13], Jxta[14], Jini[15] などが挙げられる．

Mozart は，並行制約に基づくマルチパラダイム言語である Oz の分散実装である．Mozart では，プロセス間通信に遅延具体化リスト (Stream) を用いたり，単一代入変数が存在する点などで KL1 と似ており，Stream 取得が，遠隔ノードへのアクセス取得となる点は，DKLIC と同じである．しかし，Mozart では Stream ではなく Stream への参照 (URL に似た形式) を遠隔ノードに渡しているため，動的に変化する分散環境においてこの参照が有効ではなくなる場合を特別に扱わなくてはならない．DKLIC では Stream (分散論理変数) を渡し，Stream が有効性を失った場合 (ソケットが閉じた場合) は即座に空リストにすることで検出できる点が異なる．

Jxta は，P2P ネットワーク上の分散資源の表現やアクセス方法などを定義したプロトコルであり，Java による実装 [16] が存在する．特に Pipe (通信路) を資源と見なし，宣伝 (advertisement) したり，advertisement (XML による記述) によって通信路の一端を lookup できる点が，分散論理変数と似ている．advertisement に有効期限を設け定期的に更新することで有効性を特別に保証している．DKLIC ではこのような定期更新のための余計なトラフィックは存在しない．しかし，[17] を見るとネットワーク上の FireWall や NAT などへの対応が十分に配慮されており，資源の参照が位置情報に依存しない URN 方式で表現される点などは今後 DKLIC でも参考にしていくべきである．

Jini は，サーバ集合体の管理，それらが提供するサービスの検索，サービスの移動を行

なう. Javaオブジェクトのネーミング, オブジェクト移送, コード移送の技術とも言える. 多くのネーミング技術に共通だが, サービス仕様の記述は手動でコーディングされる. これはサーバの振舞との致命的な乖離を誘発する. DKLICは, サービス仕様の記述方法に静的解析結果を用いることを目標としている. プログラム (最終的にはコンパイラ) によってサービス仕様とサーバの振舞との一致が保証される点が大きく異なる. しかし, DKLICにおけるプロセス移送, コード移送の実装の参考になると考えられる.

## 5.11 まとめと今後の課題

### 5.11.1 まとめ

本論文では, KLICの分散拡張であると同時にネットワーク透過環境の提供と分散アプリケーションに必要な機能を提供するようなKL1分散処理系DKLICを提案した. ソケット上に通常のKL1論理変数とほぼ等価な分散論理変数を実装した. これにより分散アプリケーション記述における低レベルなソケット通信の隠蔽だけでなく, TCPソケット通信上でプロセス間通信路の端点を表現し, 移送することが可能になった. これは, 単一プロセスによるKL1アプリケーションと分散アプリケーションとの相違がなくなることを意味している.

また, この分散論理変数に対する名前付けと解決を行なうネーミングプロセスを実装した. これは, サーバ仕様によって問い合わせることで, サーバプロセスへアクセスするための通信路 (分散論理変数) を取得できる仕組みを実装したことを意味する. さらに, ネーミングプロセス同士の自動接続を実装し, 複数ネーミングプロセスが連携して名前解決を行なう事で, ネーミングプロセスによるP2Pネットワーク構築と自動拡張を実現した. これによりホスト数やノード数が動的に増する不均一な分散環境上にネットワーク透過なプログラミング環境を提供することに成功した.

### 5.11.2 今後の課題

**中継排除の実装** 課題として中継排除プロトコルの実装が挙げられる. これを実装することで, 主に複数ネーミングプロセスによる名前解決によって発生する中継者を排除することを実現できる.

**複雑なネットワーク上での動作** IPマルチキャストによる他のネーミングプロセスとの自動接続を実装したが, IPマルチキャストがFireWall, NATなどのネットワーク境界を越えられないために, フラットなネットワーク上でしか動作しない. Jxta[17] などではRandevous Peerというネットワーク境界でルーティングを行なうノードを提案している. DKLICにもこのような仕組みが必要と考えられる.

**KL1実行モデルの完全な分散実装** 遠隔述語呼出, プロセス移送, コード移送の実装も挙げられる. これらを実装すると, 分散した各ゴールプールを一つの大きなゴールプールと見なすことが可能になり, KL1の実行モデルの分散環境上への完全実装を実現できる.

つまり, プロセスやルールが分散環境上に散らばりそれらが一つのKL1プロセスであるかのように見える.

**静的解析利用によるプロトコル保証**　KL1論理変数に対する入出力モード解析[18]は, 入出力モードだけでなく, あるプログラムによるプロセスが処理可能な通信プロトコル, 呼び出せるメソッドを静的に解析できる. 解析結果をネーミングのサーバ仕様に用いることで, より正確な仕様記述を可能にできる.

一方で, 本論文で実装した分散論理変数により, 接続後のサーバクライアントは単一のKL1プログラムによるプロセスであると見なせるようになった. つまり, クライアントとサーバの解析結果を併合し, 接続前に矛盾がないかを解析することが容易になった. 無矛盾な場合のみ接続することで, ネーミングプロセスがプロトコルに関しての性質を保証することが可能になる.

**例外処理の導入**　現在のKLICはKL1レベルでの例外処理をサポートしていない. 通信障害などのエラーに対して脆弱である. KLIC例外処理[19]に関する研究を利用することで対処できると考えられる. また, 不完全なKLIC処理系内部での例外処理を再点検する必要がある.

## 5.12　サンプルプログラム

### 5.12.1　ネーミングプロセス呼出

```
:- module main.

main :- true| ns:nameserver.
```

### 5.12.2　サーバ

```
:-module main.
main :- true |
      ns:register([name(intlist_3)],R),
      server(R)@lower_priority.

server(Ls) :-
      Ls = [listen(Res)|Ls2],
      server2(Res,Ls2).
server2(Res,Ls) :- Res=normal((A,B,C)) |
      intlist(A,B,C),    % intlist 呼び出し部分
      Ls = [listen(Res2)|Ls2],
      server2(Res2,Ls2).
intlist(N,Max,List):- N > Max | List = [].
```

```
intlist(N,Max,List):- N =<Max |
      List = [N|List2],
      N1 := N + 1,
      intlist(N1,Max,List2).
```

### 5.12.3 クライアント

```
:- module main.
main :- true |
      ns:lookup([name(intlist_3)],R),
      proxy(R).
proxy(R) :- R = abnormal | builtin:print(not_found).
proxy(R) :- R = normal(S) |
      S = (1,10,List),
      sum(List,0,Res),
      io:outstream([print(Res),nl]).
sum(List,Sm,Res):- List=[]        | Res = Sm.
sum(List,Sm,Res):- List=[X|List2] |
      Sm2 := X + Sm,
      sum(List2,Sm2,Res).
```

## 5.13 ジェネリックオブジェクトの書き方

ジェネリックオブジェクトは, 以下の骨組みを持たなければならない. 自前の関数はどこにおいても構わないが (C コンパイラに怒られなければ), その他はこのとおりの順番でなければならない.

```
/*************************************************************************
   これは dataObject の場合, generator, consumer の場合も適切なファイル
   を読み込まなければならない.
*************************************************************************/
#include <klic/gdobject.h>
#include <klic/gd_macro.h>

/*************************************************************************
   自分の構造を宣言する部分. method_table は必ず書かなくてはならない.
   さらに独自のメンバを持つ場合は, ここに記述すればよい.
*************************************************************************/
GD_OBJ_TYPE {
  struct data_object_method_table* method_table;
};
```

```
/*************************************************************************
   独自関数の定義．ただし，便利なKLICマクロの多くは，メソッド定義マク
   ロに隠れた引数を使っている．例えば g_allocp(ヒープ割り付けポイント)
   などである．従って隠れた引数を関数引数に受け継がなければならない．
   詳細は gd_macro.h を見るとよい．引数の変化は呼出側変数への反映が必要
   である．また，declare_globals によってKLICの大域変数を利用できる．
*************************************************************************/
static function1(q* g_allocp, q arg1, int i){}

/*************************************************************************
   メソッド定義．'_' の後に引数個数を書く．KL1プログラムでの引数個数
   から1差し引いた値を書く．メソッド名に'_' があるなら'_' でエスケープ．
   a_b(Obj,A,B) ならば，a__b_2 となる．メソッドは必ず GD_RETURN によっ
   てリターンしなくてはならない． G_STD_DECL によって様々な大域変数や
   メソッド(マクロ)を使う事ができる．
*************************************************************************/
GDDEF_METHOD(methodA_2){
  G_STD_DECL;
   /** 略 **/
  GD_RETURN;
}
GDDEF_METHOD(methodB_5){}
GDDEF_METHOD(methodC_1){}

/*************************************************************************
   独自メソッドを定義するなら GDUSE_MY_GENERIC を，独自GCを定義するなら
   GDUSE_MY_GC を定義しなくてはならない．
*************************************************************************/
#define GDUSE_MY_GENERIC
#define GDUSE_MY_GC

/*************************************************************************
   GCは必ず，GD_RETURN_FROM_GC(newself) によってリターンしなければ
   ならない．newself は新領域にコピーされた自分自身へのポインタである．
   新領域へのコピーは，GDSET_NEWOBJ_IN_NEWGEN(newself) によって得る．
*************************************************************************/
GDDEF_GC(){
  G_STD_DECL;
  GD_OBJ_TYPE *newself;
  GDSET_NEWOBJ_IN_NEWGEN(newself);
```

```
  GD_RETURN_FROM_GC(newself);
}

/*************************************************************************
   前に定義した述語を並べる.
*************************************************************************/
GDDEF_GENERIC()
{
  G_STD_DECL;
  GD_SWITCH_ON_METHOD{
    GD_METHOD_CASE(methodA_2);
    GD_METHOD_CASE(methodB_5);
    GD_METHOD_CASE(methodC_1);
    GD_METHOD_CASE_DEFAULT;
  }
  GD_RETURN;
}

/*************************************************************************
   オブジェクトのnewメソッド. 必ず, GD_RETURN_FROM_NEW(newself) に
   よってリターンしなければならない. newself は,
   GDSET_NEWOBJ_FOR_NEW(newself,GD_OBJ_SIZE(newself)) によって得る.
   GD_STD_DECL_FOR_NEW は, G_STD_DECL と同様である.
*************************************************************************/
#include <klic/gd_methtab.h>
GDDEF_NEW(){
  GD_STD_DECL_FOR_NEW;
  GD_OBJ_TYPE* newself;
  GDSET_NEWOBJ_FOR_NEW(newself,GD_OBJ_SIZE(newself));
    /** 略 **/
  GD_RETURN_FROM_NEW(newself);
}
```

generator や consumer オブジェクトの場合は特有のメソッドを定義しなければならないことがある (例えば generate メソッドなど) が, ここでは触れない.

## 参考文献


[1] 五十嵐宏：分散 KL1 言語処理系の設計と実装, 卒業論文, 早稲田大学理工学部情報学科 (1999).

[2] 高山啓：並行論理型言語 KL1 による分散 KL1 言語処理系の実装, 卒業論文, 早稲田大学理工学部情報学科 (2001).

[3] 高木祐介：KL1 による分散 KL1 言語処理系の実装, 卒業論文, 早稲田大学理工学部情報学科 (2000).

[4] 高木祐介：DKLIC 処理系における分散論理変数の資源管理, 修士論文, 早稲田大学大学院理工学研究科 (2002).

[5] 松村量：並行論理型言語 KL1 の分散拡張のための遠隔ノード管理の実装, 卒業論文, 早稲田大学理工学部情報学科 (2001).

[6] 粉川友宏：DKLIC 処理系における分散資源の位置管理, 卒業論文, 早稲田大学理工学部情報学科 (2002).

[7] 五十嵐宏：メタインタプリタに基づく分散並行論理型言語処理系, 修士論文, 早稲田大学大学院理工学研究科 (2001).

[8] Ueda, K.: A Pure Meta-interpreter for Flat GHC, a Concurrent Constraint Language, in *Lecture Notes in Computer Science*, Vol. 2407 of *LNAI*, pp. 138–161, Springer-Verlag (2002).

[9] Nakashima, H. and Inamura, Y.：An Efficient Messaage Transfer Mechanism By passing Transit Processors, in *Proc. Joint Symposium on Parallel Processing*, pp. 123–130, 情報処理学会 (1992).

[10] 高山啓, 松村量, 高木祐介, 加藤紀夫, 上田和紀：分散言語処理系 DKLIC の設計と実装, 日本ソフトウェア科学会第 19 回大会論文集 (2002).

[11] 関田大吾：*Inside KLIC Version 1.0*, KLIC Task Group AITEC/JIPDEC (1998).

[12] Roy, P. V., Haridi, S., Brand, P., Smolka, G., Mehl, M. and Scheidhauer, R.: Mobile Objects in Distributed Oz, *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 5, pp. 804–851 (1997).

[13] Haridi, S., Roy, P. V., Brand, P. and Schulte, C.: Programming Languages for Distributed Applications, *New Generation Computing*, Vol. 16, No. 3, pp. 223–261 (1998).

[14] Verbeke, J., Nadgir, N., Ruetsch, G. and Sharapov, I.: Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment, in *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA*, Vol. 2536 of *Lecture Notes in Computer Science*, Springer (2002).

[15] Sun microsystems, Inc: Jini アーキテクチャの仕様 (1999).

[16] Sun microsystems, Inc: *Technical Shell Overview* (2001).

[17] Traversat, B., Abdelaziz, M., Duigou, M., Hugly, J.-C., Pouyoul, E. and Yeager, B.: *Project JXTA Virtual Network*, Sun microsystems, Inc (2002).

[18] Ueda, K. and Morita, M.: Moded Flat GHC and Its Message Oriented Implementation Technique, *New Generation Computing*, Vol. 11, pp. 3–34 (1993).

[19] 金木祐介, 加藤紀夫, 上田和紀：KLIC 処理系における UNIX プロセス間通信を利用した例外処理の実装, 日本ソフトウェア科学会第 6 回プログラミングおよび応用のシステムに関するワークショップ (SPA2003) (2003).

# 第6章　例外処理機構の設計と実装

並行論理型言語 KL1 は GHC (Gaurded Horn Clauses) に基づいたプログラミング言語で、並行処理に必要な複雑な同期処理を自動化しているという特徴がある。もともと、KL1 は並列推論マシン (PIM) 用の言語だが、KL1 プログラムを C 言語にコンパイルし、汎用の計算機、つまり UNIX 環境上でも動作できるようにしたのが KLIC 処理系 [1, 2, 3] である。

KL1 の特徴である、同期処理の自動化は並行処理だけではなく分散処理などにも有用である。しかし、現在の KLIC 処理系には例外処理機構が無いため、大規模な分散アプリケーションを構築するためには実用的とは言えない。本研究では UNIX プロセスを利用して、KLIC 処理系に例外処理機構を実装した。これにより分散環境での例外処理機構や、フォールトトラレンスを意識したアプリケーションのためのミドルウェアの構築が目指せるようになった。

## 6.1　はじめに

本研究は並行論理型言語 KL1 の処理系 KLIC に例外処理機構を実装することを目的とする。

現在の KLIC 処理系は、UNIX シグナルの割り込みや KL1 固有の例外 (永久中断、ゴール実行の失敗、単一化の失敗) などが起こると KLIC ランタイム自体が終了してしまい、それを回避する手段は用意されていない。そこで、本研究では KLIC 処理系に UNIX プロセスを利用して例外処理機構を実装し、例外を捕捉、報告することに成功した。これによって、分散環境での例外処理機構や、フォールトトラレンスを意識したアプリケーションのためのミドルウェアの構築が目指せるようになった。

本論文は本節を含めて 8 節で構成されている。6.2 節では、例外処理の書式について述べる。6.3 節から 6.5 節にかけては、本例外処理機構の概要と実装の詳細を述べ、6.6 節は本例外処理機構のプログラム例を、6.7 節ではまとめ、そして 6.8 節では今後の課題について述べる。

## 6.2　例外処理の書式

ここでは、並行論理型言語 KL1 の言語特性を考えた例外処理の書式について考察する。書式について考えることは、例外処理の対象範囲をどう指定するかという問題も同時に含んでいる。まず、先に並行論理型言語 KL1 について簡単に触れる。

### 6.2.1 並行論理型言語 KL1

並行論理型言語 KL1 は GHC に基づいた一階述語論理のプログラミング言語である。そのプログラムの実行は他の手続き型言語とは大きく違っていて、項書き換え言語である Prolog に近い。

KL1 プログラムは、ガードつき節の集合からなる。ガードつき節とは下記のような形をしている。

$$h :- G \mid B.$$

ここで、$h$ を節のヘッド、$G$ をガード、$B$ をボディという。

KL1 プログラムは、ゴールプール中のゴールの書き換えを繰り返すことで実行が進む。ゴールはヘッドにマッチする時、ボディで指定されたゴール列に書き換えられる。ガードは節の適用条件である。適用できる節が無いゴールは適用出来るようになるまでサスペンドする。ゴールプールが空になると実行は終了する。ゴール同士は論理変数によって通信して協調動作している。

### 6.2.2 他の言語との比較

例外処理は、古くは BASIC の時代から存在していて、現在では JAVA をはじめとして多くの言語がその機能を持つ。BASIC の例外処理は非構造化例外処理といわれ、行番号やラベルなどで例外発生時の処理を制御していた。現在の JAVA 処理系などが採用している try-catch 構文は構造化例外処理といわれ、現在の主流なプログラミング概念にそったものになっている。

しかし、KL1 ではプログラムソース上のゴールの位置関係と実行順序はまったく関係がないため、JAVA、BASIC などの例外処理の書式のように例外処理対象を行単位で指定することはできない。

Prolog 処理系の中には、失敗などの例外が起こるタイミングで呼び出されるゴールを指定する例外処理機構が組み込まれているものある。

### 6.2.3 例外処理機構のモデル

本研究の例外処理のモデルは、並列推論マシン PIM 上の KL1 処理系 [4] に実装されていた荘園モジュールを参考にしている。荘園は KL1 プログラムの実行のある部分 (荘園トップレベルのゴールと、そのゴールから派生した子孫ゴール群) を内部にもつ空間である。また、荘園の中にまた荘園を作ることができ、荘園は一般に木構造となる。荘園は実行制御、資源管理、例外処理の機能を備えていた。本例外処理機構を実装するにあたって荘園の書式、例外処理の機能などを参考にしている。

以下に荘園の書式を示す。

$$\texttt{shoen:execute}(Code, ArgV, M0, M1, Exc, C, Rep)$$

$Code$ は生成する荘園のトップレベルのゴールの述語名で $ArgV$ はそのゴールの引数、$Exc$ は処理する例外事象を決めるマスクパターンを指定する。そして $Rep$ は実行状態の報告で例外の報告はここに流れる。他の引数は本研究では参照しないので省略する。

荘園はゴールを指定することで、その荘園の適用範囲をそのゴールから派生する子孫ゴール群全てにしている。本例外処理系でも 6.3.2 節で述べるように、荘園と同じ指定方法で例外対象範囲を指定することにする。

## 6.3 例外処理機構の動作

### 6.3.1 例外処理機構の概要

現在の KLIC 処理系は、一つのゴールの実行失敗がランタイム全体を終了させてしまう。これを回避するのが本例外処理機構の一つの大きな目標である。本研究では UNIX プロセスを利用することによりこの問題を回避する。KLIC 処理系自体に大きく手をいれる方法もあったが、修正の量が膨大になるため本研究ではより実装が容易な方法を選んでいる。したがって KLIC 処理系にできるだけ手をいれないように実装することが一つのポリシーとなっている。

また、UNIX プロセスを利用することにより KLIC ランタイム自体が終了しないようにするのではなく、例外を発生して KLIC ランタイムが終了するという状況を逆に利用することが出来る。詳細は、6.4 節で述べる。

この例外処理機構の基本的な考え方は、例外処理対象のゴール群を別の UNIX プロセスの KLIC ランタイム上で実行させることによって、例外処理対象ゴール群をその他のゴールから切り離してしまうことにある。したがって、例外処理対象のゴール群が例外を発生して、その KLIC ランタイムが終了しても、その他のゴールがある KLIC ランタイムは存続する。

また、この別プロセスにするという特徴を活かして、子プロセスが終了する時の終了コードに例外情報を流すという方法で例外の捕捉と報告を実現している。詳細は、6.5 節で述べる。

### 6.3.2 例外処理機構の書式

本例外処理機構が提供する述語を例外処理モジュールと呼ぶ。例外処理モジュールは通常の述語を呼び出すのと同じように呼び出される。書式を以下に示す。

`exception:execute(`*GOAL*`,`*EXCEPTION*`)`

引数 *GOAL* には *Module:Predicate(Arg,...)* という書式で例外対象にすべきゴール群のトップレベルのゴールを指定する。つまり、そのゴールから派生する子孫ゴール群が全て例外処理の範囲に入ることになる。

ゴールの引数には具体化される方向 (モード) があり、6.5.3 節で詳しく述べるように、例外処理時に通信路を閉じるためには変数のモード情報が必要になる。そこで、この例外処理モジュールの中に値が入って行く論理変数には + を、出て来る論理変数には - を明示的に付ける。実際にはモードが切り替わる時のみ明示的に付ければ良い。

引数 *EXCEPTION* は例外処理報告用の論理変数で、`err(`$N$`,`$C$`,`$M$`,`$P$`)` という項で具体化され、以下の情報を報告する。

- $N$: normal か abnormal が返る。プロセス間の論理変数による通信が正常に終了したかどうか。
- $C$: エラーコード、整数値
- $M$: エラーメッセージ、文字列、正常終了なら NORMAL が返る。
- $P$: 子プロセスの ID、整数値

以下に使用例を示す。

```
exception:execute(main:test(+(A),-(B)),E)
```

この例では、test/2 を呼び出すゴールとそのゴールから派生するゴール群を例外処理の対象として実行する。test/2 が正常に終了、または例外を発生して終了するとその報告が $E$ に具体化される。

### 6.3.3 例外処理機構の実行の流れ

以下に例外処理機構の実行の流れを示す。

1. 例外対象となるゴールを引数にした例外処理モジュールが呼ばれる。
2. 自分のプロセスのコピーである別の UNIX プロセス (子プロセス) を fork によって生成し、KLIC ランタイムを起動する。
3. 指定されたゴールとそのゴールから派生する子孫ゴール群をその別プロセス上の KLIC ランタイム上で実行する。
4. 子プロセス上のゴール群が正常終了もしくは、例外発生などで異常終了したらその情報を親プロセスに渡す。
5. 異常終了によって引き起こされたゴール間の通信障害を復旧する。具体的には通信路を閉じる (通信路内の未定義変数を具体化する) ことにより永久中断を起こすなどの障害を防ぐ。

## 6.4 UNIX プロセスによる例外処理

### 6.4.1 UNIX プロセスの生成

新しい UNIX プロセスを生成し KLIC ランタイムをその上で動作させるために、KLIC の組込み述語である fork_with_pipes を使用する。この述語は、内部で fork を使用していて、自分のプロセスのコピーを別の UNIX プロセス上に作成することができる。そのため、KLIC ランタイムごと別プロセスにコピーされるので、KLIC ランタイムをあえて実行する必要はなく、そのまま fork_with_pipes の返り値でプログラムを分岐することで、親プロセス、子プロセス上でそれぞれの動作をさせることができる。

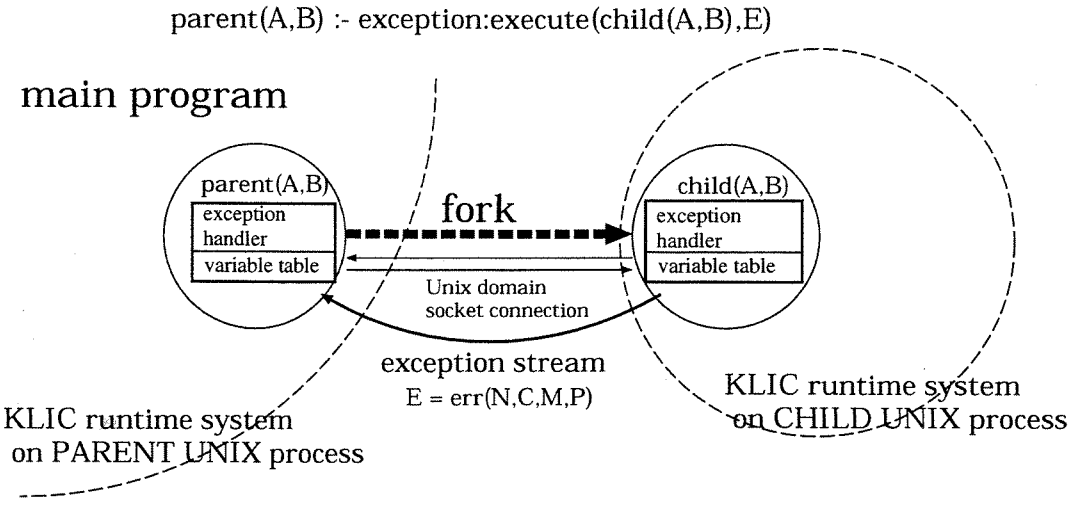



図 6.1: 例外処理システム 構成図

ここで、一つの問題が生じる。前述したように自分のコピーを作成するため、コピーが始まる前にサスペンドまたはエンキューされたゴールは当然、子プロセスでもサスペンドまたはエンキューされた状態になっている。本来の動作では、子プロセスでは例外対象のゴール以外は実行させたくないので、これらの余計なゴールを削除する必要がある。

そこで例外処理モジュールが呼ばれるとすぐに子プロセスを作成し、ゴールプール内の例外処理対象ゴールを実行する処理に必要なゴールを除くサスペンド、エンキューしている全てのゴールをゴールプールから削除する。

### 6.4.2 プロセス間通信と論理変数

KL1 はゴール同士が論理変数による通信によって協調動作しながら実行が進む。本例外処理機構では例外処理対象のゴールとそのゴールから派生する子孫ゴール群を、例外処理モジュールが作った UNIX プロセス上 (子プロセス) で実行する。そして、例外処理モジュールを呼び出している親プロセスと例外処理対象の子プロセス全体の実行は、通常の単一プロセス上での実行と同じ振舞いをしなければならない。つまり、子プロセス、親プロセスという二つの UNIX プロセス上のゴール間に論理変数による通信路を張る必要がある。

ここで、KL1 の分散処理系 DKLIC[5] の研究成果を利用する。DKLIC は分散論理変数表と 2 本のソケットを利用して異なるノード間に分散論理変数を提供し、ネットワーク透過な分散プログラミングを提供するミドルウェアである。本研究ではこの分散論理変数表を元にプロセス間論理変数表を実装した。プロセス間論理変数表については 6.4.3 節で詳しく述べる。

ソケットを TCP で張るとそのポート番号管理が非常に複雑になる。そこでソケットはポートの重複などを容易に避けるため UNIX ドメインのソケットを使用する。UNIX ドメインには “eXi” および “eXo” という文字列に子プロセスの PID を付加したものを使

用する。例えば、"eXi0031335" と "eXi0031335" である。PID は同じ UNIX 上ならプロセスに一つの番号であるので、そのドメイン名が重複しないように管理する必要が無くなる。

### 6.4.3 プロセス間論理変数表

プロセス間論理変数表は DKLIC の分散論理変数表の一部に対して修正を加えたものである。修正は非常に微小であるので本論文では触れない。ここではプロセス間論理変数表の元になっている分散論理変数表について概要を述べる。

並行論理型言語 KL1 の変数は単一代入であり、未定義状態を持つ論理変数である。ゴール間通信には頭から徐々に具体化されるリストが利用されることが多い。しかし、KLIC 処理系には異なるノード間でも論理変数によって通信できるようなライブラリは実装されていなかった。

そこで、DKLIC は異なるノード間にソケットを張り、分散論理変数表を介して論理変数による通信を実現する。つまり、ノード A とノード B が論理変数 $X$ による通信をするとする。まずノード A で論理変数 $X$ が分散論理変数表に ID と共に登録される。そうするとノード B へその登録状態が送信されノード B でも ID と共に登録される。そして、変数表はその変数 $X$ の具体化を監視していて、もしノード A で $X$ が具体化されたならノード B にその変数 $X$ が具体化されたこと知らせ、ID と具体値を送る。そうすると、ノード B でも変数 $X$ が具体化されることになる。

以上のように分散論理変数表は、ノード間で共有される論理変数の ID を管理し、その変数の具体化を監視する。

## 6.5 例外の捕捉と報告の仕組み

本例外処理機構は、UNIX プロセスを作成しその上で別の KLIC ランタイムを起動することのより、そのランタイム上の例外が他の KLIC ランタイム上にあるゴール群に影響を及ぼすのを防ぐ他に、例外を捕捉、報告することも可能にしている。

fork を使用して作成された子プロセスには、wait という関数を発行できる。この wait というのは、子プロセスの終了を親プロセスが調べたり、待ったりする時などに使用される。この wait 関数にはさらに子の終了状態を得る機能があり、主に、exit の引数、または UNIX シグナルの値を wait は受け取る。つまり、wait を発行することによって、ある子プロセス上で実行されていた KLIC ランタイムの終了状態を知ることができる。

UNIX シグナル発生によって KLIC ランタイムが終了している場合には、そのまま wait 関数がシグナル番号を受け取ることができる。しかし、KLIC 特有の永久中断などの例外は、シグナル番号で区別できないため、そのままでは例外の種類を判別することができない。

そこで、exit 関数の引数を利用して KL1 固有の例外を判別する。つまり、例えば永久中断 (ゴール同士がお互いに論理変数による通信を待ち続けデッドロックを起こす状態) が起こるときには、永久中断を検知し KLIC ランタイムを終了させようとする部分がかならずあるので、KLIC ランタイムのその部分を修正して、exit 関数によって終了する

ようにし、さらにその引数に発生した例外特有の番号をとるようにづる。実際にはシグナル発生以外の例外はある一つの関数で処理されているので、そこに例外の種類を整数に変換したものを引数に与え、exit 関数を発行するような修正を加える。

以上のようにして、KLIC 特有の例外は exit 関数の引数によって、UNIX シグナルは手を加えることなく wait 関数によって捕捉出来るようにした。

この方式の利点としてユーザが例外を定義できることが挙げられる。本例外処理機構は、exit 関数の引数に与えた番号をそのまま報告するようにしている。したがって、例外処理内で起動しているゴールを exit 関数で終了するようにし、その引数に適当な整数を与えればそのままその数字を例外報告として受けとることができる。

### 6.5.1 wait のタイミング

wait 関数は基本的に子の実行が終了するまで、wait 関数を呼び出したプログラムの実行をブロックする。本例外処理機構の場合、プログラムの本流が例外処理対象ゴールの終了を待つことになる。そのため、子プロセスを作成してすぐに wait を発行すると、その時点で親の全動作が停止するのでプロセス間論理変数表も止まってしまい、そのせいで子プロセスも通信待ち状態になり終了しなくなりデッドロックを起こす。

そこでどのタイミングで wait を発行するかというと、前述したように親プロセスが止まって困るのはプロセス間論理変数表が動作しなくなるからであるので、それの終了を待って発行することにする。なお、この場合でも子プロセス側が通信が終わった後に永久ループするようなプログラムでは永久に wait() することになってしまう状態が発生するが、それはプログラマの責任にまかせる。この問題は、タイムアウト機能を付け、ある時間が経つと子プロセスを強制終了をするという方法で回避できるが、現在は実装していない。

### 6.5.2 捕捉後の処理

KL1 は複数のゴールが論理変数を介した通信によって協調動作して実行が進んで行く。ところが、例外が発生してあるゴールの実行が止まってしまうと当然そのゴールとの通信自体も止まってしまう。したがって、もし例外を発生して死んでしまったゴールからの通信を待っているようなゴールがいると、永久にそのゴールは実行を中断することになり、永久中断の原因になってしまう。

例えば、図6.2のように、ゴール A、B、C がありそれらが論理変数による通信路 (channel) によって結ばれている時に、例外処理システム上のゴール C が例外を発生し終了してしまう状況を考える。プロセス C は例外処理システム上で終了するので、ゴール A、ゴール B は動作し続ける。しかし、ゴール A がゴール C からの通信を受け取って動作している場合には永久にその通信をゴール A は待ってしまうことになる。これでは、例外処理対象側は処理できても、その他のゴールに影響を及ぼしてしまう。

したがって、そのような論理変数の通信路にたいしてなんらかの処理をする必要がある。ここで、二つの方法が考えられる。一つは、ゴール C を復旧してゴール A からの通信路と再接続させることである。もう一つは、その通信路を閉じるという方法である。後

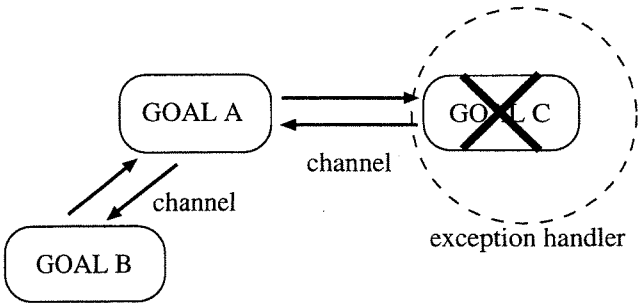



図 6.2: ゴール間通信と永久中断

者の方式では当然、例外によって通信路が閉じたことをゴール A が検知し、それに対する処理を持っていることが望まれる。

例外発生ゴールの復帰は、スナップショットをどのようにとるか、およびゴール実行の整合性などの複雑な問題を含んでいるため、本例外処理機構ではゴールの復帰によって通信路の後処理をすることは避けた。したがって、ゴールの復帰に関してはプログラマに委ねることにする。

### 6.5.3 通信路の閉鎖

ここでは論理変数による通信路をどうやって閉じるかについて述べる。

本例外処理機構の場合、問題となるゴール間の通信にはプロセス間論理変数表が仲介をしている。つまり、そのゴール間で具体化していない論理変数はプロセス論理変数表が管理しているので、例外が発生したら変数表をその未定義変数をなんらかの値 (例えば `exception`) で具体化してしまえばその通信路は閉じることになる。

しかし、全ての未定義変数を具体化すればよいわけではなく、例外処理対象から出て行く通信が問題になっているのでその通信路だけを閉じる。逆に入って来る通信を閉じようとすると、他のゴールからの具体化と衝突が起きてシステム自体が単一化失敗の例外を起こしてしまう。

論理変数の通信の方向は動的に判別することができないので、現状では 6.3.2 節で述べたように論理変数の通信の方向、つまりモード情報をあらかじめつけて変数表モジュールに知らせるようにしている。

## 6.6 プログラム例

以下に本例外処理機構を使用したプログラム例を示す。

```
main :-
 B=[divide(3,1),divide(1,0),divide(8,2)],
 calc(B).

calc([]).
```

```
calc([A|B]) :-
 io:outstream([print(R),nl]),
 exception:execute(main:div(+(A),-(R)),E),
 calc(B).

div(divide(X,Y),R) :-
 R:=X/Y.
```

このプログラムは単純に割算をさせ、表示させるプログラムである。calc/1 に divide($X$,$Y$) という命令をリストで与えると $X$ 割る $Y$ をする。div/2 が割算をし結果を返す述語で、これを例外処理対象とする。

本来なら divide(1,0) という命令は 1 割る 0 となり、ゼロ除算なので例外を発生して KLIC ランタイム自体が終了し、その後の divide(8,2) は実行されない。しかし、本例外処理機構を上記のプログラミング例のようにして使用すると、割算の例外を処理し、その後の処理 divide(8,2) を続行することができる。

以下に動作環境を挙げる。

- klic-3.003-extio-shared
  http://www.ueda.info.waseda.ac.jp/~takagi/kl1/
- fork_with_pipes を使用するために runtime/gunix.kl1 の改変
- KLIC 特有の例外情報を得るために runtime/trace.c の改変

## 6.7 まとめ

本研究では並行論理型言語 KL1 の処理系 KLIC に例外処理機構を実装した。本例外処理系は、異なる UNIX プロセスに例外処理対象となるゴール群を隔離することで、例外発生による影響がその他のゴールに及ぶことを防いだ。そして、UNIX シグナルの割り込みや KL1 固有の例外 (永久中断、ゴール実行の失敗、単一化の失敗など) を捕捉し、その例外を報告することができるようにした。これにより分散環境での例外処理機構や、フォールトトラレンスを意識したアプリケーションのためのミドルウェアの構築が目指せるようになった。

## 6.8 今後の課題

### 6.8.1 例外処理後の処理

6.5 節でも述べたが、例外処理後の処理についてはまだ完成されたものになっていない。今回はゴールの復帰をユーザにまかせ、またゴールが接続していた通信路に関しても閉じるという方法で解決をした。このままでもユーザが通信の内容を知っていて、さらに通信路を別に保持するような形で拡張すれば、その通信路の回復ゴールの再起動などができるようにプログラミングすることができるであろうが、できるなら例外処理機構側でそのようなものを実装し、ユーザーにはそれを選べる形で提供したい。

### 6.8.2 アプリケーションの開発

本研究は、KLIC 処理系には無かった例外処理機構を実装し、例外を捕捉、報告することができるようにした。しかし、大規模なアプリケーションや、例外処理が多数使用される時などにどう動作するかの検証はまだである。また、その時本例外処理機構が記述性の高いものであるか、分散処理への応用に耐えうるものであるのかなどの検証などは今後の課題である。

## 参考文献


[1] 関田大吾, Inside KLIC Version 1.0. *KLIC Task Group*, AITEC/JIPDEC, 1998.

[2] http://www.klic.org/

[3] 瀧和男 編, 第 5 世代コンピュータの並列処理. bit 別冊, 共立出版, 1993.

[4] ICOT PIMOS 開発グループ, PIMOS マニュアル (第 3.0 版), 1991.

[5] 松村 量, 高山 啓, 高木 祐介, 加藤 紀夫, 上田 和紀, dklic: KL1 による分散 KL1 言語処理系の実装, 日本ソフトウェア科学会第 19 回論文集, 2002.

# 第7章　Flat GHCの純インタプリタの構築

This paper discusses the construction of a meta-interpreter of Flat GHC, one of the simplest and earliest concurrent constraint languages.

Meta-interpretation has a long history in logic programming, and has been applied extensively to building programming systems, adding functionalities, modifying operational semantics and evaluation strategies, and so on. Our objective, in contrast, is to design the pair of (i) a representation of programs suitable for code mobility and (ii) a pure interpreter (or virtual machine) of the represented code, bearing networked applications of concurrent constraint programming in mind. This is more challenging than it might seem; indeed, meta-interpreters of many programming languages achieved their objectives by adding small primitives into the languages and exploiting their functionalities. A meta-interpreter in a pure, simple concurrent language is useful because it is fully amenable to theoretical support including partial evaluation.

After a number of trials and errors, we have arrived at *treecode*, a ground-term representation of Flat GHC programs that can be easily interpreted, transmitted over the network, and converted back to the original syntax. The paper describes how the interpreter works, where the subtleties lie, and what its design implies. It also describes how the interpreter, given the treecode of a program, is partially evaluated to the original program by the unfold/fold transformation system for Flat GHC.

## 7.1 Introduction

### 7.1.1 Meta-Interpreter Technology

Meta-interpreter technology has enjoyed excellent affinity to logic programming since the seminal work by Bowen and Kowalski [5]. It provides us with a concise way of building programming systems on top of another. This is particularly useful for AI applications in which flexibility in designing and modifying inference mechanisms is of crucial importance. Interactive programming environments such as debuggers or visualizers are another example in which interpreters can play important rôles. Extensive survey of meta-interpretation in logic programming can be found in [11], Chapter 8.

Critics complain of performance degradation incurred by the interpreter technology, but the speed of system prototyping with interpreters and symbolic languages cannot be matched by any other methodologies. Hardwiring all design choices into a lower-level language such as C may be done, but at the latest possible stage and to the least

extent. Indeed, due to Java and scripting languages, interpreter technologies – including bytecode interpreters and its optimization techniques such as just-in-time compilers – are now quite ubiquitous outside the world of symbolic languages. Java demonstrated that poor initial performance of non-optimized interpreters was acceptable once people believed that the language and the system design as a whole were the right way to go.

### 7.1.2 Concurrency and Logic Programming

The *raison d'être* and the challenge of symbolic languages are to construct highly sophisticated software which would be too complicated or unmanageable if written in other languages. Logic programming has found and addressed a number of such fields [4]. While many of those fields such as databases, constraints, machine learning, natural languages, etc., are more or less related to Artificial Intelligence, concurrency seems special in the sense that, although somewhat related to AI through agent technologies, its principal connection is to distributed and parallel computing.

Distributed and parallel computing is becoming extremely important because virtually all computers in the world are going to be interconnected. However, we have not yet agreed upon a standard formalism or a standard language to deal with concurrency. Due to the lack of appropriate tools with which to develop networked applications, computers communicate and cooperate much more poorly than they possibly can.

Concurrent logic programming was born in early 1980's from the process interpretation of logic programs [34]. Relational Language [7], the first concrete proposal of a concurrent logic language, was followed by a succession of proposals, namely Concurrent Prolog [20], PARLOG [8] and Guarded Horn Clauses (GHC) [27]. KL1 [29], the Kernel Language of the Fifth Generation Computer Systems (FGCS) project [22], was designed based on GHC by featuring (among others) *mapping* constructs for concurrent processes. To be precise, KL1 is based on Flat GHC [28], a subset of GHC that restricts guard goals to calls to test predicates.

The mathematical theory of these languages came later in the generalized setting of concurrent constraint programming (CCP) [18] based on Maher's logical interpretation of synchronization [12]. Grand challenges of concurrent logic/constraint programming are proposed in [32].

Although not as widely recognized as it used to be, Concurrent Prolog was the first simple high-level language that featured channel mobility exactly in the sense of $\pi$-calculus [15]. When the author proposed GHC as an alternative to Concurrent Prolog and PARLOG, the principal design guideline was to retain channel mobility and evolving process structures [22], because GHC was supposed to be the basis of KL1, a language in which to describe operating systems of Parallel Inference Machines as well as various knowledge-based systems. The readers are referred to [22] for various researchers' personal perspectives of the FGCS project.

### 7.1.3 Meta-Interpretation and Concurrency

Another guideline of the design of GHC was the ability to describe its own meta-interpreter. Use of simple meta-interpreters as a core technology of system development was inspired by [5], and early work on Concurrent Prolog pursued this idea in building logic-based operating systems [21].

A key technology accompanying meta-interpretation turned out to be partial evaluation. Partial evaluation of a meta-interpreter with an additional "flavor" with respect to a user program will result in a user program with the additional "flavor" that runs almost as efficiently as the original user program [24].

This idea, though very elegant, has not become as popular as we had expected.

One reason is that before the booming of the Internet, a program ran either on a single processor or on parallel processors with a more or less uniform structure, where a hardwired approach was manageable and worked. However, software for distributed computing environments is much harder to build, configure and reconfigure, and run persistently. Such software would not be manageable without a coherent solution to the difficulties incurred by heterogeneous architectures, process and code mobility, and persistence.

Another reason is that the languages and the underlying theories were not mature enough to allow full development of the idea. Meta-interpreters of many programming languages achieved their objectives by adding small primitives into the language and exploiting their functionalities. Those primitives were often beyond the basic computational models of the languages. We believe that *pure* symbolic languages are the right way to go in the long run, because only with theoretical support we can expect a real breakthrough.

### 7.1.4 Goal of This Paper

In this paper, we discuss how we can construct a meta-interpreter of Flat GHC, one of the simplest and earliest concurrent constraint languages. Our objective is to design the pair of

1. a representation of programs suitable for code mobility and interpretation, and
2. a pure, simple interpreter of the represented code.

One of the motivations of the work is to use concurrent logic/constraint programming as a concise tool for networked applications. There are strong reasons to choose concurrent logic/constraint programming as a framework of distributed computing.

First, it features channel mobility, evolving process structures, and incomplete messages (messages with reply boxes), all essential for object-based concurrent programming.

Second, it is unlike most other concurrency frameworks in that data structures (lists, trees, arrays, etc.) come from the very beginning. This means that there is little gap

between a theoretical model and a practical language. Actually, a lot of applications have been written in concurrent logic/constraint languages, notably in KL1 and Oz [23].

Third, it has been extremely stable for more than 15 years. After GHC was proposed, the main variation was whether to feature *atomic tell* (publication of bindings upon commitment) or *eventual tell* (publication after commitment). However, by now both concurrent logic programming and concurrent constraint programming seem to converge on *eventual tell*, the simpler alternative [22][26]. Indeed, concurrent constraint programming with *ask* and *eventual tell* can be thought of as an abstract model of Flat GHC.

Last, as opposed to other parallel programming languages, it achieves clear separation of concurrency (concerned with logical aspects of programs) and parallelism (concerned with physical mapping of processes). We regard this separation of concerns as the most important achievement of KL1 and its parallel implementation [29]. In other words, by using logical variables as communication channels we had achieved 100% network transparency within system-area networks (SAN). The fact that programs developed and tested on sequential machines ran at least correctly on parallel machines has benefited us enormously in the development of parallel software. We believe that this feature should be explored in distributed software as well.

Addressing networked applications using interpreters as a core technology is promising because flexibility to cope with heterogeneity is more important than performance. However, it is not obvious whether we can write a reasonably simple interpreter in a pure concurrent logic/constraint language such as Flat GHC. A meta-interpreter in a pure, simple concurrent language is fully amenable to theoretical support including partial evaluation and verification. Also, it can help *analytic approach* to language design [32], because meta-interpretation is considered an acid test of the expressive power of the language. The rôle of an interpreter technology in networked applications should be clear since an interpreter is just another name of a virtual machine.

## 7.2 Previous Work

Meta-interpreters of symbolic languages date back to a Lisp interpreter in Lisp around 1960 [13]. Prolog interpreters in Prolog were available and widely used in 1970's; an example is the interpreter of the *de facto* standard DEC-10 Prolog.

Meta-interpreters of Concurrent Prolog can be found in various papers. Figure 1 shows two versions, the first one in [20] and the second in [17].

Program (a) is very similar to a Prolog interpreter in Prolog, but it relies on the "large" built-in primitive, `clause/2` (`clause` with two arguments), that performs synchronization, evaluation of clause guards, and committed choice. The only thing reified by the interpreter is parallel conjunction. Program (b) takes both a program and a goal as arguments, and reifies the unification of the goal with clause heads and the

```
reduce(true).
reduce((A,B)) :- reduce(A?), reduce(B?).
reduce(A) :- A\=true, A\=(_,_) | clause(A?,B), reduce(B?).
```

(a) Without a program argument

```
reduce(Program,true).
reduce(Program,(A,B)) :-
    reduce(Program?, A?), reduce(Program?, B?).
reduce(Program,Goal) :-
    Goal\=true, Goal\=(A,B),
    clause(Goal?,Program?,Body) |
    reduce(Program?,Body?).
clause(Goal,[C|Cs],B) :-
    new_copy(C?,(H,G,B)), Goal=H, G | true.
clause(Goal,[C|Cs],B) :-
    clause(Goal,Cs?,B) | true.
```

(b) With an explicit program argument

Fig. 7.1: Meta-Interpreters of Concurrent Prolog

evaluation of guards. Note, however, that most of the important operations are called from and performed in clause guards. In particular, `clause/3` calls itself recursively from within a clause guard, forming a *nested (or deep) guard.*

While Concurrent Prolog employed read-only annotations as a synchronization primitive, GHC replaced it with the rule that no bindings (constraints) can be published from the guard (including the head) of a clause to the caller of the clause.

Figure 2 shows a GHC interpreter in GHC in [27]. Here it is assumed that a built-in predicate `clauses/2` returns in a *frozen* form [16] a list of all clauses whose heads are potentially unifiable with the given goal. Each frozen clause is a ground term in which original variables are indicated by special constant symbols, and it is *melted* in the guard of the first clause of `resolve/3` by `melt_new/2`. The goal `melt_new(C, (A :- G|B2))` creates a new term (say $T$) from a frozen term $C$ by giving a new variable for each frozen variable in $C$, and tries to unify $T$ with `(A :- G|B2)`. However, this unification cannot instantiate `A` because it occurs in the head of `resolve/3`.

The predicate `resolve/3` tests the candidate clauses and returns the body of arbitrary one of the clauses whose guards have been successfully solved. This many-to-one arbitration is realized by the multi-level binary clause selection using the nested guard of the predicate `resolve/3`. It is essential that each candidate clause is melted after it has been brought into the guard of the first clause of `resolve/3`. If it were melted before passed into the guard, all variables in it would be protected against instantia-

```
call(true  ) :- true | true.
call((A, B)) :- true | call(A), call(B).
call(A     ) :- clauses(A, Clauses) |
    resolve(A, Clauses, Body), call(Body).

resolve(A, [C|Cs], B) :- melt_new(C, (A :- G|B2)), call(G) | B=B2.
resolve(A, [C|Cs], B) :- resolve(A, Cs, B2) | B=B2.
```

Fig. 7.2: Meta-Interpreter of GHC

tion from the guard. We must protect variables accessible from outside but allow local variables to be instantiated.

Again, this GHC meta-interpreter calls `resolve/3` from within a guard recursively. However, our lesson is that, except for meta-interpreters, we can dispense with general nested guards. To put it more precisely, we can dispense with guard goals that may instantiate local variables; restricting guard goals to calls to *test* predicates is a more realistic choice. Test predicates are predicates defined in terms of clauses with no body goals. A nice property of test predicates is that they deterministically succeed or fail depending on their arguments. They are regarded as specifying conditions, as opposed to predicates for specifying concurrent processes. Test predicates defined using guarded clauses may call themselves recursively from guards, but unlike general nested guards, there is no need to maintain multiple layers of variable protection to implement synchronization. In this sense, languages with restriction to test predicates have been called *flat* languages. In most implementations of flat languages, test predicates are further restricted to predefined ones.

Later development of concurrent logic languages can be phrased as *devolution as evolution* [26][32] in the sense that it focused on high-performance, compiler-based implementation of flat languages. Strand [9], KL1 and Janus [19] all belong to this category. Accordingly, there was less work on meta-interpreters for the last 10 years. Huntbach [11] shows a meta-interpreter that implements *ask* using `match/2`, a special primitive discussed in detail in Sect. 7.3.3. Although using `match/2` to implement *ask* is a natural idea, `match/2` turns out to have properties not enjoyed by other goals definable in concurrent logic languages. This motivated us to design a meta-interpreter that does not use `match/2`.

Distributed computing based on concurrent constraint programming is not a new idea. The Oz group has done a lot of work in this direction [10]. However, code mobility in Oz is based on bytecode technology, and Oz has added to CCP a number of new constructs including ports (for many-to-one communication), cells (value containers that allow destructive update), computation space (encapsulated store, somewhat affected by nested guards of full GHC and KL1's *shoen*), and higher-order. This is in sharp contrast with the minimalist approach taken in this paper.

## 7.3 The Problem Statement

Now let us state the goal and the constraints of our problem precisely. Our goal is to design a binary Flat GHC predicate, say `exec`, that

- takes
  1. a multiset $G$ of goals (represented as a list) to be executed and
  2. a ground representation of the program $P$ to execute $G$, and
- behaves exactly like $G$ running under the ordinary compiled code for $P$.

The predicate `exec`/2 is sometimes called a *universal* predicate because it can be tailored, at run time, to whatever predicate you like.

The only built-in primitives the `exec`/2 program is allowed to use are those definable using (a possible infinite number of) guarded clauses. Other primitives are considered extralogical and are ruled out. Observing this constraint will enable the resulting interpreter to run on KLIC [6], which is in our context considered as a (Flat) GHC-to-C compiler and its runtime system. Flat GHC and KLIC carefully rule out extralogical built-in primitives because they can potentially hamper efficient implementation and theoretical support.

A solution to the problem is not obvious because Flat GHC and KLIC do not have general nested guards, on which the interpreter of full GHC in Sect. 7.2 depends in a fundamental way.

Some remarks and discussions on our requirements are in order, which are (1) representation of code, (2) representation of runtime configuration, and (3) primitives for *ask* (matching) and *tell* (unification).

### 7.3.1 Representation of Code

Meta-interpreters vary in the representation of programs. Some retrieve programs from the internal database using primitives like `clause`/2. This is not suited to our goal of code mobility and persistence. Some use a list of clauses in which variables are represented using variables at the level of the interpreters. This is considered misuse of variables, as criticized by later work on meta-programming, because those variables are improperly scoped and awkward to handle. One solution is to use a higher-order construct as in Lambda Prolog [14], and another solution is to come up with a ground representation of variables. Although the higher-order approach gives us the most natural solution, the difference between the two solutions is not large when the programs to be represented have no nested scope, which is the case with Prolog and Flat GHC.

As we will see later, we have chosen to represent a variable in terms of a reserved unary constructor with an integer argument. This could be viewed as a de Bruijn notation as well.

### 7.3.2 Representation of Runtime Configuration

In a rule-based language where programs (rewrite rules) are given separately from expressions (goals), how to represent runtime configurations and how to represent the programs are independent issues. The two alternatives for the representation of runtime configurations are

1. to reify logical variables and substitutions and handle them explicitly, and
2. not to reify them but use those at the interpreter level.

We adopt the latter, because

- an interpreted process must be *open-ended*, that is, it must be able to communicate with other native processes running in parallel with the interpreter,
- the reification approach would therefore require 'up' and 'down' predicates to move between the two levels of representation and (accordingly) a full-fledged meta-programming framework in the language, and
- explicit representation can cause performance degradation unless elaborate optimization is made.

### 7.3.3 Primitives for Matching/Ask and Unification/Tell

In the CCP terminology, Prolog and constraint logic languages in their basic forms are *tell*-only languages because unification or constraint solving is the attempt to publish bindings (constraints) to the binding environment (constraint store). In contrast, concurrent logic/constraint languages are *ask*+*tell* languages which additionally feature matching (in algebraic terms) or the asking of whether a given constraint is entailed (in logical terms) by the current store. So how to implement *ask* and *tell* in an interpreter is a key design issue.

The Prolog and GHC versions of *tell* are unification over finite trees and can be written as `unify`$(G, H)$ or $G = H$. This has the following properties:

1. *Immediate* — It either succeeds or fails and does not suspend.
2. *Monotonic* — Its success/failure can depend on the current store; that is, `unify`$(G, H)$ that succeeds under some store can fail under a store augmented with additional constraints. However, if we consider failure as a over-constrained store, `unify`$(G, H)$ can be thought of as an operator that monotonically augments the current store.
3. *Deterministic* — The conjunction of all *tell*s generated in the course of program execution deterministically defines the current store.

Now we consider the properties of *ask*, which appears in concurrent logic languages as matching between a goal and a clause head. Let $\sigma$ be the current store under which the *ask* is performed. We suppose `match`$(G, H)$

- *succeeds* when there exists a substitution $\theta$ such that $G\sigma = H\sigma\theta$,
- *suspends* when there is no such $\theta$ but $G\sigma$ and $H\sigma$ are unifiable, and
- *fails* when $G\sigma$ and $H\sigma$ are non-unifiable.

Clearly, `match`$(G, H)$ is not immediate. Furthermore, it is neither monotonic nor deterministic with respect to suspension behavior:

- `match(X,Y)` will succeed when `Y` is uninstantiated but may suspend when `Y` is instantiated. This behavior is opposite to that of ordinary CCP processes which can never be suspended by providing more constraints.
- `match(X,Y)` $\wedge$ `match(3,Y)` under the empty store succeeds if executed from left to right but suspends if executed from right to left.

When simulating matching between a goal $G$ and a clause head $H$ using `match/2`, $H$ must have been renamed using fresh variables, and $H$ is therefore immune to $\sigma$. If this convention is enforced, `match/2` enjoys monotonicity, that is, if `match/2` succeeds under $\sigma$, it succeeds under $\sigma\sigma'$ for any $\sigma'$. The convention guarantees determinism as well.

The lesson here is that the scope of the variables in $H$, the second argument of `match/2`, should be handled properly for `match/2` to enjoy reasonable properties. As suggested by [12], the proper semantics of `match`$(G, H)$ would be whether $\sigma$ interpreted as an equality theory *implies* $G = \exists H$. Thus the second argument should specify an existential closure $\exists H$ rather than $H$. However, then, the second argument would lose the capability to *receive* matching terms from $G$. For instance, the recursive clause of `append/3` in GHC is

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

while the CCP version of the above clause would be less structured:

```
append(X0,Y,Z0) :- ask(∃A,X(X0=[A|X])) |
    tell(X0=[A|X]), tell(Z0=[A|Z]), append(X,Y,Z).
```

To summarize, while implementing *tell* in an interpreter is straightforward, implementing *ask* without introducing new primitives is a major design issue.

## 7.4 A Treecode Representation

In this section, we discuss the design of our treecode representation of Flat GHC programs, which is interpreted by the treecode interpreter described in the Sect. 7.5.

### 7.4.1 Treecode

Treecode is intermediate code in the form of a first-order ground term which is quite close to the original source code. It is more abstract and "structured" than ordinary bytecode sequences that use forward branching to represent `if...then...else`. Trees are much more versatile than sequences and are much easier to represent and handle than directed graphs. Indeed, the booming of XML tells us that standard representation of tagged trees has been long-awaited by a great number of applications, and XML trees are little more than first-order ground terms.

Of course, the control flow of a program forms a directed graph in general and we must represent it somehow. Directed graphs could be created rather easily by unification over rational terms, but we chose to dispense with circular structures by representing recursive calls (that form circularity) using explicit predicate names. When the interpreter encounters a predicate call, it obtains the code for the predicate using an appropriate lookup method. An optimizing interpreter may create a directed graph by "instantiating" each predicate call to its code before starting interpretation.

An alternative representation closer to source code is a set of rewrite rules. However, it turns out that a set (represented as a list) of rewrite rules is less suitable for interpretation. This is because GHC "bundles" predicate calls, synchronization and choice in a single construct, namely guarded clauses. While this bundling simplifies the syntax and the semantics of Flat GHC and captures the essence of concurrent logic programming, guards – even flat guards – can specify arbitrary complex conditions that may involve both conjunctive and disjunctive sets of multiple synchronization points. Programmers also find it sometimes cumbersome to describe everything using guarded clauses exactly for the reason why Prolog programmers find that the `(P -> Q ; R)` construct sometimes shortens their programs considerably.

As we will see soon, treecode still looks like a set of clauses, but the major difference from a set of clauses is that the former breaks a set of guards down to a tree of one-at-a-time conditional branching. In this sense, treecode can be regarded as *structured intermediate code.*

### 7.4.2 Treecode By Example

Now we are in a position to explain how treecode looks like. Throughout this section we use `append/3` as an example. The treecode for `append/3` is:

```
treecode(6,
  [c(1=[], b([<(2)= <(3)],[])),
   c(1=[>(4)|>(5)],
          b([<(3)=[<(4)|>(6)]],[append(5,2,6)]))])
```

The first argument, 6, stands for the number of variables used in the treecode, and the second argument is the main part of the treecode.

$$
\begin{array}{rcl}
\langle treecode \rangle & ::= & \langle casecode \rangle \mid \langle bodycode \rangle \\
\langle casecode \rangle & ::= & \text{list of } \langle choice \rangle\text{'s} \\
\langle choice \rangle & ::= & \texttt{c(}\langle ask \rangle\texttt{, }\langle treecode \rangle\texttt{)} \\
\langle ask \rangle & ::= & \langle reg \rangle \texttt{ = } \langle term \rangle \mid \langle reg \rangle \langle relop \rangle \langle term \rangle \\
\langle bodycode \rangle & ::= & \texttt{b(}\langle tells \rangle\texttt{, }\langle goals \rangle\texttt{)} \\
\langle tells \rangle & ::= & \text{list of } \langle tell \rangle\text{'s} \\
\langle tell \rangle & ::= & \langle annotatedreg \rangle \texttt{ = } \langle term \rangle \mid \langle annotatedreg \rangle \texttt{ := } \langle term \rangle \\
\langle goals \rangle & ::= & \text{list of } \langle goal \rangle\text{'s} \\
\langle goal \rangle & ::= & \langle pred \rangle\texttt{(}\langle reg \rangle\texttt{,...)} \\
\langle annotatedreg \rangle & ::= & \texttt{[}\langle annotation \rangle\texttt{]}\langle reg \rangle \\
\langle annotation \rangle & ::= & \texttt{<} \mid \texttt{<<} \mid \texttt{>} \\
\langle reg \rangle & ::= & \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \ldots \\
\langle term \rangle & ::= & \langle functor \rangle\texttt{(}\langle annotatedreg \rangle\texttt{, ...)} \\
\langle relop \rangle & ::= & \texttt{>} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{=<} \mid \texttt{=:=} \mid \texttt{=\=}
\end{array}
$$

Fig. 7.3: Syntax of Treecode

The readers may be able to guess what it does basically, since it is quite similar to the original source code:

```
append(X, Y,Z ) :- X=[] | Y=Z.
append(X0,Y,Z0) :- X0=[A|X] | Z0=[A|Z], append(X,Y,Z).
```

In this simple example, the treecode still looks like a list of clauses, with heads (with mutually disjoint variables) omitted and variables represented by positive integers. The constructor `c/2` forms a case branch by taking an *ask* and another treecode as arguments. The list of case branches forms a *casecode*.

The constructor `b/2` forms a *bodycode* by taking a list of *tell*s and a list of calls to user-defined predicates. The former is understood by the interpreter, while the latter involves code lookup.

A *treecode* is either a casecode or a bodycode. Figure 7.3 shows the syntax of treecode.

### 7.4.3 Representing and Managing Logical Variables

The unary constructors '<' and '>' have two purposes. First, they distinguish integer representation of variables from integer constants in the program to be interpreted. Second, they tell whether a variable has occurred before and whether it will occur later. *Initial mode*, denoted, '>', means the creation of a new variable, while *final mode*, denoted '<', means the final access to an already created variable. In `append/3`, each variable occurs exactly twice, which means that all accesses are either initial or final accesses. For variables that are read more than once, we use another reserved

unary constructor, '<<', to indicate that they are accessed in *intermediate mode*, that is, they are neither the first nor the last occurrences.

The first occurrence of a variable in each case branch (1 in the case of `append`/3) and the arguments of user-defined predicates are supposed to be final-mode. These are the only places where mode annotations are omitted for ease of interpretation.

Representing variables by positive integers suggests the use of arrays to represent them. We use a constructor g/$n$ to represent goal records, where $n$ is the number of variables in the treecode that works on the goal. The structure g/$n$ can be regarded as a *register vector* as well.

Let $a$ be the arity of the predicate represented by the treecode. The first $a$th arguments of g/$n$ are the arguments of the original goal, while the remaining arguments are local variables of the original goal. Thus this structure can be regarded both (i) as a concretization of goals that makes housekeeping explicit and (ii) as an abstraction of implementation-level goal records. When the structure is created, the first $a$th arguments are initialized to the arguments of the original goal, while the remaining arguments are initialized to the constant 0. The value of $a$ is not recorded in the treecode itself. It is the responsibility of the predicate `try`/3 to "apply" treecode to a goal record, as will be described in Sect. 7.5.

The distinction between initial, intermediate and final modes not only makes interpretation easier but also allows the reuse of the same register for different variables. For example, the code for `append`/3 could be written alternatively as:

```
[c(1=[], b([<(2)= <(3)],[])),
 c(1=[>(4)|>(1)],
        b([<(3)=[<(4)|>(3)]],[append(1,2,3)]))]
```

because

- Variable 1 in the second branch, holding the first argument of the caller, will not be accessed after its principal constructor has been known, and
- Variable 3 in the second branch, holding the third argument of the caller, will not be accessed after it has been instantiated to an non-empty list.

This is register allocation optimization which is optional in our treecode. Without it, different numbers represent different single-assignment variables and the code is more declarative. With it, the size of goal records can be reduced.

## 7.5 Structure of the Treecode Interpreter

This section describes, step by step, how our treecode interpreter works on a goal record. We focus on basic *ask* and *tell* operations. The actual interpreter handles arithmetic built-in predicates for comparison (guard) and assignment (body), but it is straightforward to include them.

The two main predicates of the interpreter are `exec/2` and `try/3`. The predicate `exec/2` takes a multiset $G$ of goals and a program $\mathcal{E}$ for executing them. We call the program an *environment* because it associates each predicate name with its treecode. The goal `exec`$(G, \mathcal{E})$ resolves predicate names in $G$ into their corresponding treecode, and invokes `try/3` for each goal in $G$ after preparing a goal record for the goal. The predicate `try/3` takes a goal record, a treecode and an environment, and applies the treecode to the goal record. The more interesting aspects of the interpreter lie in `try/3`.

### 7.5.1 Deterministic and Nondeterministic Choice

When the treecode given to `try/3` is *casecode*, it deterministically chooses one branch as follows: It picks up the first case branch of the form `c`$(Ask, Treecode)$, where $Ask$ is of the form $n = T$. This causes the interpreter to wait for the principal constructor of the $n$th argument, and when it is available, it is matched against the constructor of $T$. The $n$'s in each case branch must be identical; thus casecode has exactly one synchronization point for all its top-level *ask*s and is therefore deterministic.

When some guard involves the asking of more than one symbol, it is compiled into nested casecode. For instance, the program

```
part(_,[],    S, L ) :- true | S=[], L=[].
part(A,[X|Xs],S0,L ) :- A>=X | S0=[X|S], part(A,Xs,S,L).
part(A,[X|Xs],S, L0) :- A< X | L0=[X|L], part(A,Xs,S,L).
```

can be compiled into:

```
[c(2=[], b([<(3)=[],<(4)=[]],[])),
 c(2=[>(5)|>(2)],
   [c(1>= <<(5), b([<(3)=[<(5)|>(3)]],[part(1,2,3,4)])),
    c(1<  <<(5), b([<(4)=[<(5)|>(4)]],[part(1,2,3,4)]))])]
```

Note that the matching of the second argument with `[X|Xs]` has been factored, as would be done by an optimizing compiler.

Nested casecode is still deterministic because it has at most one synchronization point (i.e., the variable on whose value the interpreter suspends) at any time. Our experience with Flat GHC/KL1 programming has shown that the majority of predicates are deterministic.

*Nondeterministic* predicates are those which contain *disjunctive* wait, namely wait for the instantiation of one of several variables. Some of the predicates people write are nondeterministic, but most of them involve binary choice only. For instance, the following stream merging program

```
merge([],Ys,Zs) :- true | Zs=Ys.
merge(Xs,[],Zs) :- true | Zs=Xs.
merge([X|Xs],Ys,Zs0) :- true | Zs0=[X|Zs], merge(Xs,Ys,Zs).
merge(Xs,[Y|Ys],Zs0) :- true | Zs0=[Y|Zs], merge(Xs,Ys,Zs).
```

has two disjunctive synchronization points, namely the principal constructor of the first argument and the principal constructor of the second argument.

In this paper we focus on binary nondeterministic choice, which is simpler to implement than general multiway choice. It can be expressed in terms of two nondeterministic branches in the interpreter. By extending our treecode in Fig. 7.3, the treecode for `merge/3` can be written as follows:

```
treecode(4,
  (1->[c(1=[], b([<(2)= <(3)],[])),
       c(1=[>(4)|>(1)], b([<(3)=[<(4)|>(3)]],[merge(1,2,3)]))])
+ (2->[c(2=[], b([<(1)= <(3)],[])),
       c(2=[>(4)|>(2)], b([<(3)=[<(4)|>(3)]],[merge(1,2,3)]))]))
```

The extended syntax of treecode is:

$$\begin{array}{rcl} \langle treecode \rangle & ::= & \langle casecode \rangle \mid \langle bodycode \rangle \mid \langle nondeterministiccode \rangle \\ \langle nondeterministiccode \rangle & ::= & (\langle reg \rangle \texttt{ -> } \langle treecode \rangle) \texttt{ + } (\langle reg \rangle \texttt{ -> } \langle treecode \rangle) \end{array}$$

where the form $(n_1 \rightarrow treecode_1)+(n_2 \rightarrow treecode_2)$ causes the goal to wait disjunctively upon variables $n_1$ and $n_2$.

### 7.5.2 Interpreting Casecode

The *ask* part of a casecode of the form $n=T$, where $T$ is a non-variable term whose arguments are all *annotatedregs*, is interpreted by the following piece of code:

```
try_one(A0,Rn=T,B,Cs,Env) :- true |
   setarg(Rn,A0,A0Rn,ARn,A), functor(A0Rn,A0RnF,A0RnN),
   functor(T,TF,TN), test_pf(A0RnF,A0RnN,TF,TN,Res),
   try_match(Res,T,A0Rn,ARn,A,B,Cs,Env).

test_pf(F1,A1,F2,A2,Res) :- F1=F2, A1=:=A2 | Res=yes(A1).
otherwise.
test_pf(F1,A1,F2,A2,Res) :- true | Res=no.

try_match(yes(N),T,A0Rn,ARn,A0,B,Cs,Env) :- true |
   ARn=0, getargs(1,N,T,A0Rn,A0,A), try(A,B,Env).
try_match(no,    T,A0Rn,ARn,A, B,Cs,Env) :- true |
   ARn=A0Rn, try(A,Cs,Env).

getargs(K,N,T,A0Rn,A0,A) :- K> N | A0=A.
getargs(K,N,T,A0Rn,A0,A) :- K=<N |
   arg(K,T,Tk), setarg(K,A0Rn,A0Rnk,0,A0Rn1),
   getputreg(Tk,A0,A0Rnk,A1),
```

```
    K1:=K+1, getargs(K1,N,T,AORn1,A1,A).

getputreg(<(Rk), AO,ARk,A) :- true | setarg(Rk,AO,ARk,0,A).
getputreg(<<(Rk),AO,ARk,A) :- true | setarg(Rk,AO,ARk,ARk,A).
getputreg(>(Rk), AO,ARk,A) :- true | setarg(Rk,AO,_,ARk,A).
```

This is almost a Prolog program with a cut in every clause. KL1's built-in predicate, setarg$(I,T,X,X',T')$, is like Prolog's arg$(I,T,X)$ except that $T'$ is bound to $T$ with its $I$th element replaced by $X'$. This is a declarative array update primitive and used extensively in the interpreter to read data from, and write data to, goal records.

The try_one/5 program first retrieves the Rnth variable in the goal record AO, binding it to AORn. Then it checks if AORn is instantiated and its principal constructor matches that of T, using functor/3 and test_pf/5. If the matching succeeds, the first clause of try_match/8 stores (by using getargs/6) the top-level arguments of AORn to the goal record AO according to the prescription template T. Then it executes the bodycode B under the updated goal record A and the environment Env. The first goal AR0=0 binds the Rnth element in A to 0; this is to explicitly discharge a pointer from the goal record to the top-level structure that has just been *ask*ed. The interpreter uses the constant 0 as a filler when some element of a goal record does not contain a meaningful value, that is, before a meaningful value is loaded or after a meaningful value is taken away.

### 7.5.3 Interpreting Bodycode

Bodycode performs *tell*s and the spawning of user-defined body goals:

```
try(AO,b(BU,BN),Env) :- true | tell(AO,BU,A), spawn(A,BN,Env).
```

The *tell*s are not only to instantiate variables passed from the caller; it is also used to prepare non-variable terms to be passed to user-defined body goals, and to unify two variables to create a shared variable between two body goals. How tell/3 manipulates data is quite similar to how getargs/6 gets data from a non-variable goal argument. A *tell* of the form $n = T$ manipulates the $n$th element of the goal record according to the template $T$:

```
tell(AO,[(Rn=T)|BU], A) :- true |
   getputreg(Rn,AO,AORn,A1), tell_one(T,AORn,A1,BU,A).
tell(AO,[],          A) :- true | A=AO.

tell_one(<(Rk),  AORn,A1,BU,A) :- true |
   getputreg(<(Rk),A1,AORn,A2), tell(A2,BU,A). /* load Rk */
tell_one(>(Rk),  AORn,A1,BU,A) :- true |
   getputreg(>(Rk),A1,AORn,A2), tell(A2,BU,A). /* store Rk */
tell_one(T,      AORn,A1,BU,A) :- integer(T) |
```

```
    A0Rn=T,  tell(A1,BU,A).
otherwise.
tell_one(T,      A0Rn,A1,BU,A) :- true |
    functor(T,F,N), new_functor(A0Rn0,F,N),
    putargs(1,N,T,A0Rn0,A0Rn,A1,A2), tell(A2,BU,A).

putargs(K,N,T,A0Rn0,A0Rn,A0,A) :- K> N | A0Rn0=A0Rn, A0=A.
putargs(K,N,T,A0Rn0,A0Rn,A0,A) :- K=<N |
    arg(K,T,Tk), setarg(K,A0Rn0,_,A0Rnk,A0Rn1),
    getputreg(Tk,A0,A0Rnk,A1),
    K1:=K+1, putargs(K1,N,T,A0Rn1,A0Rn,A1,A).
```

Note that the two functionalities of Prolog's `functor/3` are provided by different KL1 built-ins, `functor/3` and `new_functor/3`. While `functor/3` suspends on the first argument and examines its principal constructor, `new_functor/3` creates a new structure with a constructor specified by the second and the third arguments. The major difference between `new_functor/3` and its Prolog counterpart is that the arguments of the structure are initialized to `0` rather than fresh, distinct variables. This is because we have found that initializing its elements to a filler constant and replacing them using `setarg/5` shows much better affinity with a static mode system that plays various important rôles [30] in concurrent logic programming. As discussed in [31], strong moding is deeply concerned with the number of access paths (or references) to each variable (or its value). It prefers variables with exactly two occurrences to those with three or more occurrences by giving the former more generic, less-constrained modes. Our `setarg/5` does not copy or discard the (direct or indirect) access paths to the elements of an array, including the element to be removed and the element with which to fill in the blank.

Linearity analysis [33] for Mode Flat GHC is more directly concerned with the number of access paths. Under reasonable conditions, it enables us to implement `setarg/5` as destructive update as long as the original structure is not shared.

Both mode and linearity systems encourage *resource-conscious programming.* Resource-conscious programming means to pay attention to the number of occurrences of each variable and to prefer variables with exactly two occurrences. This is not so restrictive as it might seem, and our static analyzer *klint* [33] and an automated debugger *kima* [2][3] support it by detecting – and even correcting – inadvertently too many or too few occurrences of the variables. Resource-conscious programs are easier to execute on a distributed platform because they can benefit more from compile-time garbage collection.

Finally, we show the definition of `spawn/3` for spawning body goals according to the bodycode and the current goal record:

```
spawn(A, []      ,Env) :- true | true.
spawn(A0,[B0|BN],Env) :- true |
```

```
    functor(B0,F,N), setargs(1,N,B0,A0,B,A),
    exec_one(B,Env), spawn(A,BN,Env).

/* registers once read are cleared */
setargs(K,N,B0,A0,B,A) :- K> N | B=B0, A=A0.
setargs(K,N,B0,A0,B,A) :- K=<N |
    setarg(K,B0,Bk,ABk,B1), setarg(Bk,A0,ABk,0,A1),
    K1 := K+1, setargs(K1,N,B1,A1,B,A).
```

Note that concurrent execution of body goals is realized by the concurrent execution of exec_one's.

### 7.5.4 Summary

Now we have almost finished the description of our interpreter. To be self-contained, here we show all the remaining predicates.

```
/* The interpreter's top-level */
exec([],Env)     :- true | true.
exec([G|Gs],Env) :- true | exec_one(G,Env), exec(Gs,Env).

exec_one(G,Env) :- true |
    retrieve(G,Env,TC), prepare_goalrec_body(G,TC,A,B),
    try(A,B,Env).

retrieve(G,Env,TC) :- true |
    functor(G,P,N), retrieve(P,N,Env,TC).
retrieve(P,N,[P/N-TC0|_],TC) :- true | TC=TC0.
otherwise.
retrieve(P,N,[_|Env],TC) :- true | retrieve(P,N,Env,TC).

prepare_goalrec_body(G0,treecode(N,B0),A,B) :- true |
    B=B0,
    functor(G0,_,Ng), new_functor(A0,g,N),
    transfer_args(1,Ng,G0,A0,_,A).

transfer_args(I,N,G0,A0,G,A) :- I> N | G=G0, A=A0.
transfer_args(I,N,G0,A0,G,A) :- I=<N |
    setarg(I,G0,Gi,0,G1), setarg(I,A0,_,Gi,A1),
    I1 := I+1, transfer_args(I1,N,G1,A1,G,A).

/* Simply a case branch based on the syntax of treecode */
```

```
try(A,[c(G,B)|Cs],Env)            :- true |
   try_one(A,G,B,Cs,Env).
try(A,(Rn1->Cs1)+(Rn2->Cs2),Env) :- true |
   try_two(A,Rn1,Cs1,Rn2,Cs2,Env).
try(A0,b(BU,BN),Env)              :- true |
   tell(A0,BU,A), spawn(A,BN,Env).

/* Binary disjunctive wait */
try_two(A0,Rn1,Cs1,Rn2,Cs2,Env) :- true |
   setarg(Rn1,A0,A0Rn1,ARn1,A1), setarg(Rn2,A1,A0Rn2,ARn2,A),
   try_two(A,A0Rn1,ARn1,A0Rn2,ARn2,Cs1,Cs2,Env).

try_two(A,A0Rn1,ARn1,A0Rn2,ARn2,Cs1,Cs2,Env) :- wait(A0Rn1) |
   ARn1=A0Rn1, ARn2=A0Rn2, append(Cs1,Cs2,Cs), try(A,Cs,Env).
try_two(A,A0Rn1,ARn1,A0Rn2,ARn2,Cs1,Cs2,Env) :- wait(A0Rn2) |
   ARn1=A0Rn1, ARn2=A0Rn2, append(Cs2,Cs1,Cs), try(A,Cs,Env).

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

The restrictions of the above interpreter and possible solutions to them are as follows:

1. Three unary constructors, '<', '>' and '<<', are reserved. This can be easily circumvented by wrapping non-variable as well as variable symbols by some constructors, but we did not do so for the readability of treecode.

2. Currently, the only built-in predicates provided (but not shown above) are those for arithmetics. However, other built-ins such as those used in the interpreter itself can be easily provided.

3. A nonlinear clause head, namely a head with repeated occurrences of a variable, cannot be compiled into treecode. Extending the interpreter to deal with nonlinear heads is straightforward and left as an exercise. However, the use of a nonlinear clause head to check the equality of arguments is discouraged, because it is the only construct that may take unbounded execution time by comparing two terms of arbitrarily large sizes. For distributed and real-time applications, it is desirable that the execution time of every primitive language construct is bounded.

4. The only construct whose support requires non-straightforward hacking on the interpreter is non-binary disjunctive wait. Since $n$-ary disjunctive wait is essentially $n$-ary arbitration, this could be supported by implementing an $n$-ary arbiter which observes variables $x_1, \ldots, x_n$ and returns an arbitrary $k$ such that $x_k$ has been instantiated.

The interpreter is not self-applicable in its present form, but the discussions above indicate that we are quite close to a self-applicable meta-interpreter. Note that the `otherwise` construct to specify default cases can be expressed implicitly using casecode because the *ask* parts of its branches are tested both deterministically and sequentially.

## 7.6 Partial Evaluation

How can one be assured that interpreted treecode behaves exactly the same as its original code?

Instead of showing a translator from Flat GHC to treecode and its correctness, here we illustrate how the treecode for `append/3` applied to our interpreter can be partially evaluated to its original Flat GHC code.

The rôle of partial evaluation in our framework is twofold. First, the receiver of treecode can figure out what Flat GHC code it represents. Second, although the interpreter itself is not directly amenable to static analysis because its behavior depends on the treecode given, the original code restored by partial evaluation is amenable to static analysis. In this way we can attach various kinds of type information (including mode and linearity) to the arguments of a goal whose behavior is determined by treecode.

For partial evaluation, we use unfold/fold transformation rules described in [28]. The rules consist of the following:

1. *Normalization* — executes unification goals in a guard and a body so that each clause reaches its unique normal form. A normal form should have no unification goals in guards, and all residual unification body goals should be to instantiate head variables of the clause.

2. *Immediate Execution* — deals with the unfolding of a non-unification body goal which does not involve synchronization. That is, the rule is applicable only when, for each clause $C$ in the program and each goal $g$ to be unfolded, either $g$ is reducible using $C$ or, for all $\sigma$, $g\sigma$ is irreducible using $C$.

3. *Case Splitting* — deals with the unfolding of non-unification body goals of a clause $C$ which may promote *ask*s from the guards of clauses used for the unfolding to the guard of $C$. The clause $C$ must not have unification body goals.

   To see how the Case Splitting of $C$ works, consider a goal $g$ that is about to be reduced using $C$. For $g$ to generate some output, at least one more reduction (of one of the body goals of $C$) is necessary because $C$ has no unification body goals. Case splitting enumerates all the possibilities of the first such reduction.

4. *Folding* — which is essentially the same as the Tamaki-Sato folding rule [25].

The major difference from the Tamaki-Sato rule set is that unfolding is split into two incomparable rules, *Immediate Execution* and *Case Splitting*, to deal with synchronization.

Let $\mathcal{E}$ be the treecode for append/3:

```
[append/3-treecode(6,
   [c(1=[], b([<(2)= <(3)],[])),
    c(1=[>(4)|>(1)], b([<(3)=[<(4)|>(3)]],[append(1,2,3)]))])]
```

To show that exec_one(append(X,Y,Z),$\mathcal{E}$) behaves the same as append(X,Y,Z) under its standard definition, let us start with a clause

```
append(X,Y,Z) :- true | exec_one(append(X,Y,Z),E).
```

and start applying Immediate Execution to its body goal. Using exec_one/2 shown in Sect. 7.5.4, we obtain

```
append(X,Y,Z) :- true |
  retrieve(append(X,Y,Z),E,TC),
  prepare_goalrec_body(append(X,Y,Z),TC,A,B), try(A,B,E).
```

With two more applications of *Immediate Execution*, first to the goal retrieve/3 and the second to the primitive functor/3, we obtain

```
append(X,Y,Z) :- true |
  P=append, N=3, retrieve(P,N,E,TC),
  prepare_goalrec_body(append(X,Y,Z),TC,A,B), try(A,B,E).
```

which can be normalized to

```
append(X,Y,Z) :- true |
  retrieve(append,3,E,TC),
  prepare_goalrec_body(append(X,Y,Z),TC,A,B), try(A,B,E).
```

With several steps of *Immediate Execution* and *Normalization*, we arrive at

```
append(X,Y,Z) :- true |
  transfer_args(1,3,append(X,Y,Z),g(0,0,0,0,0,0),_,A),
  try(A,[c(1=[],b([<(2)= <(3)],[])),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)]))],E).
```

where transfer_args/6 "loads" the arguments X, Y, Z to the goal record and returns the result to A. Further steps of *Immediate Execution* and *Normalization* lead us to

```
append(X,Y,Z) :- true |
  functor(X,AORnF,AORnN),
  test_pf(AORnF,AORnN,[],0,Res),
  try_match(Res,[],X,ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
```

This is the first point at which we can't apply *Immediate Execution* or *Normalization*.

We regard the primitive functor/3 as comprising clauses such as:

```
functor([],   F,N) :- true | F=[],  N=0.
functor([_|_],F,N) :- true | F='.', N=2.
functor(f(_), F,N) :- true | F=f,   N=1.
```

There is one such clause for each constructor available, but without loss of generality we can focus on the above three clauses, of which the third one is meant to be a representative of all constructors irrelevant to the current example.

Now we apply *Case Splitting* and obtain the following:

```
append([],Y,Z) :- true |
  AORnF=[], AORnN=0,
  test_pf(AORnF,AORnN,[],0,Res),
  try_match(Res,[],[],ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
append([H|T],Y,Z) :- true |
  AORnF='.', AORnN=2,
  test_pf(AORnF,AORnN,[],0,Res),
  try_match(Res,[],[H|T],ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
append(f(X),Y,Z) :- true |
  AORnF=f, AORnN=1,
  test_pf(AORnF,AORnN,[],0,Res),
  try_match(Res,[],f(X),ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
```

That is, we unfold `functor/3` and promote its *ask*s to the guards of `append/3`. The *Case Splitting* rule dictates that we should unfold `test_pf/5` and `try_match/7` as well; however, unfolding `test_pf/5` using its first clause, for instance, would promote two *ask*s, `AORnF=[]` and `AORnN=:=0`, which can never be satisfied because the two variables don't occur in the head of `append/3`. Clauses with unsatisfiable *ask*s are deleted finally. Note that clauses below the `otherwise` directive (such as the second clause of `test_pf/5`) implicitly perform all *ask*s in the clauses above the `otherwise`.

Now we come back to applying *Normalization* and *Immediate Execution*, which leads us via

```
append([],Y,Z) :- true |
  try_match(yes(0),[],[],ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
append([H|T],Y,Z) :- true |
  try_match(no,[],[H|T],ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
append(f(X),Y,Z) :- true |
  try_match(no,[],f(X),ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
```

to the following:

```
append([],Y,Z) :- true |
  getargs(1,0,[],[],g(0,Y,Z,0,0,0),A),
  try(A,b([<(2)= <(3)],[]),E).
append([H|T],Y,Z) :- true |
  getargs(1,2,[>(4)|>(1)],[H|T],g(0,Y,Z,0,0,0),A),
  try(A,b([<(3)=[<(4)|>(3)]],[append(1,2,3)]),E).
append(f(X),Y,Z) :- true | try(g(f(X),Y,Z,0,0,0),[],E).
```

Here, the rules as stated in [28] do not allow *Immediate Execution* of the third clause because we cannot form any unfolded clause to replace it. However, a close look at the reason why assures us that this clause *can* indeed be removed. The removal of a clause $C$ whose body goal can never proceed changes the behavior of a goal $g$ when there is another clause $C'$ that can reduce $g$. However, the three clauses of append/3 above don't overlap with one another; that is, any goal that can be reduced using the third clause and then gets stuck will get stuck without it.

By steps of *Immediate Execution*, we can "load" necessary values to registers:

```
append([],Y,Z) :- true |
  try(g(0,Y,Z,0,0,0),b([<(2)= <(3)],[]),E).
append([H|T],Y,Z) :- true |
  try(g(T,Y,Z,H,0,0),b([<(3)=[<(4)|>(3)]],[append(1,2,3)]),E).
```

Now we have restored the guards of the original append/3, which is much more than halfway to our goal. It remains to restore the bodies, and this can be done by repetitive application of *Immediate Execution* and *Normalization*:

```
append([],Y,Z) :- true | Y=Z.
append([H|T],Y,Z) :- true |
  Z=[H|A0Rn], exec_one(append(T,Y,A0Rn),E).
```

Finally, we fold the body goal of the second clause using the clause we coined initially, and obtain the following:

```
append([],   Y,Z) :- true | Y=Z.
append([H|T],Y,Z) :- true | Z=[H|A0Rn], append(T,Y,A0Rn).
```

We anticipate that the significance of partial evaluation in our context is it enables us to use available tools for "just-in-time" static analysis. For faster execution, designing an optimizing compiler from treecode to machine code would be more appropriate than going back from treecode to Flat GHC source code.

## 7.7 Conclusions

We have described an interpreter of Flat GHC treecode in Flat GHC. The interpreter uses only *pure* built-in primitives, that is, those whose behavior can be defined using a set of guarded clauses (e.g., `functor/3`, `setarg/5`, etc.) or by simple source-to-source transformation (`otherwise`). The interpreter is only 39 clauses long (without arithmetics), and runs directly on KLIC.

Treecode is very close to source code but is designed so that it can be easily interpreted, transmitted over the network, and stored in files. The major differences from most bytecode representations are that it is more structured and, more importantly, that it is inherently concurrent.

The design of an interpreter involves decisions as to what are reified and what are not. To allow interpreted processes to freely communicate with non-interpreted, native processes, we made the following design choices:

- *Reified*: code, reduction, concurrency and nondeterminism; goal records, argument registers and temporary registers; control structures
- *Not reified*: logical variables and substitutions (constraints); heaps; representation of terms.

Although our initial objective was to have an 100% pure interpreter of Flat GHC, the outcome can be viewed also as a virtual machine working on register vectors. The three annotations, '`>`', '`<`', and '`<<`', are reminiscent of the distinction between `put` and `get` instructions in the Warren Abstract Machine [1].

Translation from source code to treecode is straightforward for most cases. For deterministic programs, its essence is to build a decision tree for clause selection. Some complication arises only when a predicate has both conjunctive and disjunctive synchronization points. The paper did not show a concrete translation algorithm, but instead illustrated how a treecode could be translated back to its source code using partial evaluation. Note that the source code could be restored because the interpreter was a *meta*-interpreter. Partial evaluation thus ensures the applicability of program analysis to interpreted code. Type analysis is important for an interpreted process to communicate with a native process running with no runtime type information. It is also important in building a stub and a skeleton of a (marshaled) logical stream laid between remote sites.

Our primary future work is to deploy those technologies to demonstrate that concurrent logic/constraint programming can act, possibly with minimal extensions, as a high-level and concise formalism for distributed programming. Another important direction is, starting with treecode, to develop an appropriate intermediate code representation for optimizing compilers. This is important for another application of concurrent languages, namely high-performance parallel computation.

## References


[1] Aït-Kaci, H., *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, MA, 1991.

[2] Ajiro, Y., Ueda, K. and Cho, K., Error-Correcting Source Code. In *Proc. Fourth Int. Conf. on Principles and Practice of Constraint Programming (CP'98)*, LNCS 1520, Springer-Verlag, Berlin, 1998, pp. 40–54.

[3] Ajiro, Y. and Ueda, K., Kima – an Automated Error Correction System for Concurrent Logic Programs. In *Proc. Fourth Int. Workshop on Automated Debugging (AADEBUG 2000)*, Ducassé, M. (ed.), 2000.
http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings.html

[4] Apt, K. R., Marek, V. W., Truszczynski M., and Warren D. S. (eds.), *The Logic Programming Paradigm: A 25-Year Perspective.* Springer-Verlag, Berlin, 1999.

[5] Bowen, K. A. and Kowalski, R. A., Amalgamating Language and Meta-Language in Logic Programming. In *Logic Programming*, Clark, K. L. and Tärnlund, S. Å. (eds.), Academic Press, London, pp. 153–172, 1982.

[6] Chikayama, T., Fujise, T. and Sekita, D., A Portable and Efficient Implementation of KL1. In *Proc. 6th Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'94)*, LNCS 844, Springer-Verlag, Berlin, 1994, pp. 25–39.

[7] Clark, K. L. and Gregory, S., A Relational Language for Parallel Programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA'81)*, ACM, 1981, pp. 171–178.

[8] Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.

[9] Foster, I. and Taylor, S., Strand: a Practical Parallel Programming Tool. In *Proc. 1989 North American Conf. on Logic Programming (NACLP'89)*, The MIT Press, Cambridge, MA, 1989, pp. 497–512.

[10] Haridi, S., Van Roy, P., Brand, P. and Schulte, C., Programming Languages for Distributed Applications. *New Generation Computing*, Vol. 16, No. 3 (1998), pp. 223–261.

[11] Huntbach, M. M., Ringwood, G. A., *Agent-Oriented Programming: From Prolog to Guarded Definite Clauses.* LNCS 1630, Springer-Verlag, Berlin, 1999.

[12] Maher, M. J., Logic Semantics for a Class of Committed-Choice Programs. In *Proc. Fourth Int. Conf. on Logic Programming (ICLP'87)*, The MIT Press, Cambridge, MA, 1987, pp. 858–876.

[13] McCarthy, J., *Lisp 1.5 Programmer's Manual.* MIT Press Cambridge, MA, 1962.

[14] Miller, D. and Nadathur, G., Higher-order Logic Programming. In *Proc. Third Int. Conf. on Logic Programming (ICLP'86)*, LNCS 225, Springer-Verlag, Berlin, 1986, pp. 448–462.

[15] Milner, R. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999.

[16] Nakashima, H., Ueda, K. and Tomura, S., What Is a Variable in Prolog? In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984 (FGCS'84)*, ICOT, Tokyo, 1984, pp. 327–332.

[17] Safra, M. and Shapiro, E. Y., Meta Interpreters for Real, In *Information Processing 86*, Kugler, H.-J. (ed.), North-Holland, Amsterdam, pp. 271–278, 1986.

[18] Saraswat, V. A. and Rinard, M., Concurrent Constraint Programming (Extended Abstract). In *Conf. Record of the Seventeenth Annual ACM Symp. on Principles of Programming Languages (POPL'90)*, ACM Press, 1990, pp. 232–245.

[19] Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conference on Logic Programming (NACLP'90)*, The MIT Press, Cambridge, MA, 1990, pp. 431–446.

[20] Shapiro, E. Y., Concurrent Prolog: A Progress Report. *IEEE Computer*, Vol. 19, No. 8 (1986), pp. 44–58.

[21] Shapiro, E. Y. (ed.), *Concurrent Prolog: Collected Papers*, Volumes I+II. The MIT Press, Cambridge, MA, 1987.

[22] Shapiro, E. Y., Warren, D. H. D., Fuchi, K., Kowalski, R. A., Furukawa, K., Ueda, K., Kahn, K. M., Chikayama, T. and Tick, E., The Fifth Generation Project: Personal Perspectives. *Comm. ACM*, Vol. 36, No. 3 (1993), pp. 46–103.

[23] Smolka, G., The Oz Programming Model. In *Computer Science Today*, van Leeuwen, J. (ed.), LNCS 1000, Springer-Verlag, Berlin, 1995, pp. 324–343.

[24] Takeuchi, A. and Furukawa, K., Partial Evaluation of Prolog Programs and Its Application to Meta Programming. In *Information Processing 86*, Kugler, H.-J. (ed.), North-Holland, Amsterdam, 1986, pp. 415–420.

[25] Tamaki, H. and Sato, T., Unfold/Fold Transformation of Logic Programs. In *Proc. Second Int. Logic Programming Conf. (ICLP'84)*, Uppsala Univ., Sweden, 1984, pp. 127–138.

[26] Tick, E. The Deevolution of Concurrent Logic Programming Languages. *J. Logic Programming*, Vol. 23, No. 2 (1995), pp. 89–123.

[27] Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming '85*, Wada, E. (ed.), LNCS 221, Springer-Verlag, Berlin, 1986, pp. 168–179.

[28] Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988 (FGCS'88)*, ICOT, Tokyo, 1988, pp. 582–591.

[29] Ueda, K. and Chikayama, T. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.

[30] Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.

[31] Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, LNCS 1068, Springer-Verlag, Berlin, 1996, pp. 134–153.

[32] Ueda, K., Concurrent Logic/Constraint Programming: The Next 10 Years. In [4], 1999, pp. 53–71.

[33] Ueda, K., Linearity Analysis of Concurrent Logic Programs. In *Proc. Int. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, Singapore, 2000, pp. 253–270.

[34] van Emden, M. H. and de Lucena Filho, G. J., Predicate Logic as a Language for Parallel Programming. In *Logic Programming*, Clark, K. L. and Tärnlund, S. -Å. (eds.), Academic Press, London, 1982, pp. 189–198.