THE DESIGN AND IMPLEMENTATION OF AN ACTOR LANGUAGE BASED ON LINEAR LOGIC

by

Clifford Sheung-Ching Tse

B.Sc., Cognitive Sciences, M.I.T., 1993

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Science and Engineering

and

Bachelor of Science in Computer Science and Engineering

and

Master of Science in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology
May 1994

Author ____
Department of Electrical Engineering and Computer Science, May 12, 1994

Certified by ____
W. J. Dally, Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Certified by ____
V. A. Saraswat, Researcher, Xerox Palo Alto Research Center
Thesis Supervisor

Accepted by ____
F. R. Morgenthaler, Chair, Departmental Committee on Graduate Students

Contents

The Design and Implementation of an Actor Language Based on Linear Logic

by

Clifford Sheung-Ching Tse

Abstract

Recent advances in computer architecture have led to a proliferation of machines made of thousands of processors, such as the T3D, CM5, Paragon, NCube, and J-Machine. These machines have tremendous potential which will allow us to tackle problems previously unmanageable even on supercomputers. Unfortunately, we have yet to develop languages to efficiently and reliably create sophisticated software for such machines. Computer languages we have today are either computationally powerful but semantically ill-understood or semantically well-understood but computationally primitive.

This thesis describes a new language called Linear Janus. The underlying theoretical foundation of Linear Janus is Girard's linear logic, Hewitt's actors and Saraswat's Concurrent Constraint Programming (CCP). By using a minimal set of primitives for creating, communicating and synchronizing processes, Linear Janus provides a simple framework for exploiting and reasoning about concurrency and indeterminacy.

Processes in Linear Janus are encapsulated in entities called agents. Agents perform autonomous tasks and interact with each other asynchronously. Communication and synchronization happen in channels, which are concurrent objects accessible by multiple agents. Agents add information to a channel using asynchronous *tell* and retrieve information using *ask* which blocks until the *ask* can be answered. Finally, by moving from classical logic to linear logic, resources and actions are modeled precisely as logic formulas, whereas computation is interpreted as controlled deductions.

The three foci of this thesis are (1) to establish the underlying foundation of Linear Janus as a calculus for reasoning about concurrency and indeterminacy, (2) to discuss the computational properties of Linear Janus as a language for structuring and expressing concurrent and indeterminate algorithms, and (3) to outline an implementation for studying issues in developing an efficient system for Linear Janus.

Thesis Advisor: Dr. V. A. Saraswat, Xerox Palo Alto Research Center
Professor W. J. Dally, MIT Artificial Intelligence Laboratory

Chapter 1

Introduction

As traditional computer architectures reach their physical limitations, only massively parallel machines can deliver the processing power required for confronting problems of the coming decades. Massively parallel machines like Cray T3D, Thinking Machine's CM5 and Intel's Paragon, which are made up of thousands of processors, have already promised performance that far supersedes that of conventional supercomputers. However, we have yet to develop a clean and lean language to harness the power of such machines.

To understand the interaction among different processes in a concurrent system, we need a vigorous yet simple conceptual model. λ-calculus has long been regarded as a pure mathematical foundation for computation. Despite its simplicity, λ-calculus provides an astonishingly powerful framework for understanding the intricacy of computation. However, λ-calculus is intrinsically sequential. Concurrent systems present a whole new array of complexity for which we have not been able to develop a solid foundation. Our current models are predominantly concerned with architectural aspects of the systems, how individual process are interconnected, and how they may interact with each other. Within such models, reasoning and understanding are always obscured by inordinate details.

In this thesis, a model of concurrency and indeterminacy is proposed via Linear Janus. Linear Janus is a concurrent language developed from insights gained from Girard's linear logic[17], Hewitt's actors[24, 4, 12] and Saraswat's CCP[50]. The computational model of Linear Janus centers around entities called agents. Agents perform autonomous tasks and interact with each other asynchronously through channels, which are objects allowing concurrent access. Channels are the basis of all communication and synchronization in Linear Janus. Agents communicate and synchronize by sending messages or posting methods to channels. Messages are sent asynchronously but a method blocks until a matching message arrives. The formal foundation of Linear Janus is based on linear logic in which resources and actions are defined by formulas and computations are interpreted as controlled deduction. The model of concurrency and indeterminacy proposed in Linear Janus is simple yet rich enough to encompass formalisms like Petri Nets[46], π-calculus[37, 38], and actors[4, 12].

1.1 Overview of the thesis

In the first part of this thesis, we describe the theoretical foundation of Linear Janus. Chapter 2 gives an overview of the motivation for this research, as well as its theoretical and historical background; in particular, actors, CCP and linear logic are discussed in relation to each other. Chapter 3 explains our design choices as well as syntax and semantics of KLJ, the kernel of Linear Janus. Chapter 4 describes the computational properties of Linear Janus through a series of examples. Chapter 5 reviews related formalisms. Finally, chapter 6 discusses the directions that Linear Janus may evolve in the future.

In the second part of the thesis, we describe a prototype implementation of Linear Janus which includes an abstract machine and a compiler. Chapter 8 presents the abstract machine architecture and instructions. In chapter 9, the prototype compiler and some optimizations are discussed. In chapter 10, the implementation of the abstract architecture is explained. Finally, some preliminary observations about the implementation of Linear Janus are discussed in chapter 11.

Chapter 2

Motivation and Background

2.1 Motivation

The development of concurrent programming languages is hardly a new enterprise. Multithreading was proposed as early as 1960's with constructs like fork/join, co-begin/co-end, critical sections and monitors, whose descendants still have significant presence in many operating systems and programming languages[1]. These ideas, however, are less applicable nowadays, at least in their primitive forms. In the current generation of concurrent computers, there are typically thousands of threads active at the same time. The complex interaction between different components makes explicit control of individual thread tedious and error prone. The lack of formal semantics also prevents complete confidence in program correctness.

A more promising approach is that of functional programming. Functional programs are side-effect free and referentially transparent. Results of a computation only depend on the inputs and not the course of execution. In theory, functional programs can run with all the potential parallelism. In reality, pure functional languages are virtually non-existent. Functional programs with Church-Rosser property, while allowing a compiler to explore possible parallelism, are incapable of dealing with reactive tasks such as I/O handling that are common in any real systems. Imperative features (references in ML[40], M-structures in Id[7]) are frequently introduced to provide expressiveness, efficiency or both. However, adding these imperative features inevitably weakens and complicates the underlying mathematical semantics.

Imperative and functional languages both project a computational model which may be classified

[1]This focus of this thesis is on concurrent MIMD machines and languages, instead of parallel SIMD machines and languages.

as process-oriented. Object-oriented languages, on the other hand, offer a model in which tasks are performed by interacting entities called objects. Concurrency arises naturally because objects may act in parallel. Additionally, objects and classes provide a very useful abstraction for expressing and structuring concurrent computations. However, for all its conceptual simplicity, vigorous theoretical models for objects are still missing. Hewitt generalized and formalized the notion of objects to autonomous entities called actors[24]. Actors communicate with each other via asynchronous message-passing and computation evolves as actors interact. By focusing on patterns of communication, actors provides a powerful model for dynamic and asynchronous computation.

A different model of concurrent computation was introduced in Concurrent Constraint Programming(CCP)[50], where computations emerge from the interactions of concurrent processes which communicate and synchronize by placing, checking and instantiating constraints on shared variables. Instead of *reading* and *writing* to a store, processes in CCP *ask* to check if a constraint is entailed by the store and *tell* to augment the store with a new constraint. Using constraints for communication and synchronization, CCP provides a framework for exploiting fine grained concurrency.

Although CCP and actors have radically different notions of communication and synchronization, there are close connections between the two. However, as will be discussed in the following, the exact connections was revealed only with the development of linear CCP[53]. Linear CCP abandoned the classical intuitionistic logic as the underlying foundation and moved onto linear logic. With linear logic, resources and computations can be defined precisely as formulas and deductions.

Linear Janus is an actor-like language within the general framework of linear CCP. By assimilating key ideas of actors and CCP, Linear Janus provides a formal model for exploring fine grained concurrent and indeterminate computation.

The rest of this chapter introduces the theoretical and historical background behind Linear Janus. Section 2.2 discusses actor computation, section 2.3 describes the principles of CCP and section 2.4 explains the failure of classical logic as a model of computation and the solutions provided by linear logic.

2.2 Actors

Actors was proposed in the 70's by Hewitt *et al.*[24] for modeling asynchronous computation in open systems. Open systems can be defined as large collections of services which interact with each other without centralized control, complete trust and full specification. An open system has the ability to *evolve* as new services are added or removed, and as connections between services are modified. Hewitt[25] described the fundamental characteristics of open systems as follows:

- *Concurrency* due to simultaneous influx of information.
- *Asynchrony* due to non-deterministic interaction between components.
- *Decentralized control* to avoid bottlenecks and increase reliability.
- *Inconsistent information* due to information sources maintained by different components.
- *Arms-length relationships* to ensure privacy and security.
- *Continuous operation* despite component failures.

In an actor system, computation is encapsulated in entities called actors. An actor system has at least two classes of actors, built-in actors and non-primitive actors. The set of built-in actors, which perform internal functions like additions, is defined by a particular implementation. Informally, a non-primitive actor P is defined by a unique address, a set of addresses, and a script. The unique address is the reference to P, commonly referred to as P's *mailbox*; the set of addresses, referred to as P's *acquaintances*, is the set of actors that P knows; and the script consists of a collection of actions to be carried out when a message arrives at P's mailbox. In particular, an actor upon receiving a message can send messages to its acquaintances and spawn new actors, including a *replacement* for itself to service the remaining or incoming messages in its mailbox.

Actor computation is inherently concurrent since actors are autonomous and affect each other only via asynchronous messages. Additionally, through communications, an actor can acquire new acquaintances and hence can change the connectivities with other components of the system. The notion of an actor is extremely robust and in a *universal* actor system everything (including numbers and messages) can be uniformly treated as actors.

The ability of actors to interact asynchronously and to evolve dynamically is the basis of actors' power. However, actors are a minimal model of asynchronous computation. In [31], actors was shown to be a special case of CCP. This leads to the question that actor computation might be obtained in the high level framework of CCP.

2.3 CCP

CCP[50] is a generalization of concurrent logic programming and constraint logic programming. The central concept underlying CCP is to replace the notion of *store-as-valuation* behind imperative programming languages with the notion of *store-as-constraint*, which naturally allows partial information to be manipulated and recorded during computation. In imperative programming, the state of computation is captured by a set of values stored in variables. As computation progresses, the variables are updated by *single-valued* assignments. Essentially, a computation may be represented by a transition of a single point in the state space. In CCP, instead of a single point, a state is better visualized as a (possibly infinite) set of points in the state space. Variables in CCP contain constraints that represent sets of admissible valuations. Computation emerges as constraints are added to refine the sets of admissible valuations.

More concretely, instead of reading and writing a store, agents *ask* to check if a constraint (such as $X \geq 3$) is entailed by the constraints within store and *tell* to add a new constraint (such as $X = 4$) to the store. Operationally, *tell* is non-blocking while *ask* suspends on the store until the store has enough information to answer (prove or disprove) the *ask*. Communication and synchronization arise naturally when multiple agents place, check and instantiate constraints in shared stores.

The cc framework is very general and various languages can be defined by choosing a particular ask-and-tell constraint system.

An intuitive way to model actors within the cc framework is to represent a mailbox M with a logical variable L equated to a multiset of messages. Sending a message m to M can then be translated to imposing the constraint $m \in L$. However, such a translation does not work correctly. Two actors may try to communicate the same message m to an actor P. Operationally, we would like P to see both copies of m; logically, this situation cannot be distinguished from one in which only one message is sent. The problem stems from the behavior of connectives, conjunction ($\wedge$) and disjunction ($\vee$), in classical logic. Since conjunction and disjunction are idempotent, $(m \in L) \wedge (m \in L)$ and $(m \in L) \vee (m \in L)$ are both logically equivalent to $(m \in L)$. Moreover, in CCP, constraints are accumulated monotonically in a store. A constraint in a store, unlike a message in a mailbox, is never removed. Hence, CCP does not have a built-in notion of resource consumption. To solve these problems, we need to move to linear logic, in which formulas can be treated precisely as computational resources.

2.4 Classical Logic and Linear Logic

To make the following discussion more precise, we introduce the *Gentzen-sequent* notation for writing logical judgments. We let P and Q range over formulas and Γ and Δ range over finite (possibly empty) sequences of formulas. A *sequent* consists of two sequences of formulas separated by a turnstile, $\vdash$. The sequent $\Gamma \vdash \Delta$ asserts that conjunction of the formulas in $\Gamma(= P_1, \ldots, P_n)$ *implies* the disjunction of formulas in $\Delta(= Q_1, \ldots, Q_n)$, i.e.:

$$P_1 \ \wedge \ldots \wedge \ P_n \quad \rightarrow \quad Q_1 \ \vee \ldots \vee \ Q_n$$

A *proof* of the sequent $\Gamma \vdash \Delta$, constructed using the proof rules, is a finite tree where the root is labeled with $\Gamma \vdash \Delta$. A proof rule states that if the set of hypotheses is true, the conclusion will be true. A proof rule consists of two parts, a set of *hypotheses* (or sometimes called premises) and a *conclusion*, written as:

$$\frac{Hypotheses}{Conclusion}$$

In logic programming, resources and actions are formulated as atomic formulas, and relationship among objects are defined by a set of axioms. A computation is a controlled deduction from the initial set of axioms. Parallelism arises in two ways during a deduction. In the first case, conjunctive branches of the derivation are expanded concurrently (and-parallelism) and in the second, disjunctive branches of the derivation are expanded concurrently (or-parallelism). Both kinds of parallelism have been exploited in (Flat) Concurrent Prolog[6], PARLOG[11], (Flat) GHC[60] and a variety of other languages.

The close correspondence between computation and deduction has been explored extensively to develop formal semantics for logic programming languages, proof theories for program verification, semantic preserving program transformations and other vigorous mathematical tools for understanding computations. Notwithstanding, classical logic is inadequate to capture realistic computational phenomena. For example, there is no logical representation for the simple chemical equation:

$$2H_2 + O_2 \quad \rightarrow \quad 2H_2O$$

The central problem is that logic formulas are treated as static and pervasive entities. In classical logic, hypotheses and conclusions can be duplicated and discarded rather arbitrarily in a derivation due to the two structure rules, contraction and weakening:

$$(Contraction) \quad \frac{\Gamma, P, P \vdash \Delta}{\Gamma, P \vdash \Delta} \quad \frac{\Gamma \vdash \Delta, P, P}{\Gamma \vdash \Delta, P}$$

$$(Weakening) \quad \frac{\Gamma \vdash \Delta}{\Gamma, P \vdash \Delta} \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, P}$$

Intuitively, contraction states that if two assumptions of a hypothesis P can prove a property, then one assumption of P can prove the same property. Weakening, on the other hand, states that if we do not need an assumption P to prove a property, then introducing P can still prove the same property. With these structural rules, there is an inevitable loss of control: if we start with two hypotheses P and $P \rightarrow Q$, then we can logically assert Q, or in our sequent representation as:

$$P, P \rightarrow Q \vdash Q$$

However, since we can duplicate P before using it to discharge the implication $P \rightarrow Q$, we can easily get both P and Q in our conclusion, i.e.:

$$P, P \rightarrow Q \vdash P \wedge Q$$

where, one copy of P is used to discharge the implication, while the other copy is retained in the conclusion. It is not difficult to see that we can in fact have arbitrary many copies of P and Q in the above conclusion. Such abilities to duplicate and discard formulas are essential in mathematical proofs, but disastrous in modeling real world computations. In computational terms, contraction allows us to use resources without ever consuming them, while weakening creates false dependencies by demanding more than what is really necessary to perform a computation. In reality, computational resources have to be accounted for carefully. With contraction and weakening, classical logic can only deal with stable truth and has no vocabulary to precisely express the dynamics of computations.

In hindsight, it is clear that the contraction and weakening must be re-examined if logic is to model interesting real world computations directly. As proposed by Girard in linear logic[17], contraction and weakening should be eliminated from the structural rules. Without contraction and weakening, each formula has to be used exactly once in a deduction. Within such a linear system, each formula can now be naturally regarded as a finite resource that cannot be duplicated and discarded without effort. Such linearity also leads to notions of *resource consumption* and *resource conservation* in linear implications, where formulas used for discharging the implications are *consumed* and the others are *conserved*.

2.5 Intuitionistic Linear Logic

The rules for the intuitionistic fragment of linear logic is as follows:

Axiom:

$$(Id) \quad \frac{}{A \vdash A}$$

Structural Rule:

$$(Exchange) \quad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}$$

Cut Rule:

$$\frac{\Gamma \vdash A \quad A, \Delta \vdash B}{\Gamma, \Delta \vdash C}$$

Logical Rules:

$$(1_R) \quad \frac{}{\vdash \mathbf{1}} \qquad\qquad \frac{\Gamma \vdash A}{\Gamma, \mathbf{1} \vdash A} \quad (1_L)$$

$$(\otimes_R) \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad\qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} \quad (\otimes_L)$$

$$(\&_R) \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \& B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \& B \vdash C} \quad (\&_L)$$

$$(\oplus_R) \quad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C} \quad (\oplus_L)$$

$$(\multimap_R) \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad\qquad \frac{\Gamma \vdash A \quad B, \Delta \vdash C}{\Gamma, A \multimap B, \Delta \vdash C} \quad (\multimap_L)$$

The multiplicative conjunction ($\otimes$) corresponds to the customary notion of “and”, whereas the additive conjunction (&) represents a choice between two actions and we have the ability to decide on the choice. The additive disjunction ($\oplus$) is similar to the additive conjunction (&) in representing a choice between two actions, except that we cannot decide on which choice. Finally, linear application ($\multimap$) represents the action which consumes a formula and produce another one.

To illustrate the operational meanings of the connectives, we use the classic vending machine example. We let A, B and C stand for the following propositions:

A : input $1

B : output a Coke

C : output a Pepsi

Suppose the vending machine is governed by two rules $A \multimap B$ and $A \multimap C$. The first rule states that a dollar will buy you a Coke and the second states that a dollar will buy you a Pepsi. With only one dollar, one can never buy both a Coke and a Pepsi, and indeed $A \multimap B \otimes C$ does not follow from the initial rules. But, we can either buy a Coke or a Pepsi; the rules imply $A \multimap B \& C$ (a choice between B and C). Additionally, the vending machine allow us to *make* the choice on buying either a Coke or a Pepsi. However, if the vending machine is designed by a linear logician, he may decide that the output of the vending machine be indeterminate by specifying $A \multimap B \oplus C$ as the machine rule. In such case, inserting a dollar will either get you a Coke or a Pepsi but you do not know which.

Using part of the above fragment, we can already derive a simple calculus for concurrent and indeterminate computations. If we represent actions and resources by logic formulas, *tensor* ($\otimes$) gives a logical reading for concurrency because formulas thus conjoined can be manipulated in parallel, whereas *with* ($\&$) gives a reading for indeterminacy because computation may proceed on one of two branches, and finally, implications are *rules of actions*. However, computations expressed within this fragment are necessarily finite since implications and formulas are consumed once they take part in any action. The solution is to introduce the storage operator *of course* ("!"), which is defined by the following logical rules:

$$(!_R) \quad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A, \Delta} \qquad \frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B} \quad (Dereliction)$$

$$(Contraction) \quad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B} \quad (Weakening)$$

In essence, the storage operator allows us to use a formula either 0, 1, or more times, i.e. $!A$ means that A can be duplicated by *contractions*, discarded by *weakening*, or restored to a linear formula by *dereliction*.

Returning to the example above, our chemical equation can be modeled precisely in linear logic as:

$$H_2 \otimes H_2 \otimes O_2 \multimap H_2O \otimes H_2O$$

The computational structure of linear logic is remarkably rich. In the following chapters, we show that by exploiting a subset of linear logic, we can develop a simple yet powerful calculus for concurrent and indeterminate computations.

2.6 Historical Background

Linear Janus was inspired by Janus[51] which was developed by Saraswat, Kahn and Levy. Janus is a CCP language designed to address the shortcomings of committed choice logic programming languages for distributed programming. A subset of Janus, called Lucy, was observed to mimic actor computation closely[31] while capable of providing a higher level of abstractions than actors. However, an exact connection between actors and CCP cannot be drawn in the classical framework due to problems mentioned above.

By moving to linear logic, Saraswat has started developing a new framework, called (logically), linear CCP. Linear CCP is a very general framework and Linear Janus is an instance of the actor subset. My involvement in the formulation of Linear Janus started in January 1992 and by then Saraswat had started working on the foundation of linear cc languages. A portion of the result described in this part of the thesis are developed from Saraswat's early notes on linear cc languages and through collaboration with him.

Chapter 3

The Language

This chapter formally introduces the kernel subset of Linear Janus. Section 3.1 examines the issues in communication and synchronization. Section 3.2 explains the notion of a Linear Janus channel which plays a key role in the whole language. Section 3.3 presents the definition of KLJ, the kernel language which forms the building block of Linear Janus. Section 3.4 formulates operational semantics for Linear Janusand the connection between operational and logical interpretations of agents. Finally, section 3.5 summarizes and discusses a number of critical issues and design choices of Linear Janus.

3.1 Overview

The principle notions of any concurrent system are those of communication and synchronization. In multithreading, mutable shared memories are frequently used to implement communication and synchronization among processes. In such models, interacting processes share common resources which allows each process to read or write for the purpose of communication and synchronization with other processes. To coordinate accesses to the shared resources, atomic constructs like mutex, semaphore, conditional variables (Modula3), and definitional variables (Strand[14] and PCN[15]) are used in many shared memory concurrent languages.

On the other hand, message-passing is more commonly used in object-based or agent-based systems. With message-passing, communication can be either synchronous (such as in CSP[10] and CML[47]) or asynchronous (such as in actors). When communication is synchronous, the communicating parties require a handshake process commonly referred to as *rendezvous*. During rendezvous, the sender initiates a request for communication and waits until the receiver accepts its request.

When both the sender and receiver are in sync, communication takes place. When communication is asynchronous, the messages have to be buffered to allow the sender to send the message without blocking. On the other end, the receiver may retrieve the message at some later time or may suspend on an empty buffer until a message arrives. The buffers can be either ordered or unordered. If the buffers are ordered, the arrival order of the messages at a buffer is preserved during retrieval. If the buffers are unordered, the receiver may see the messages in an unspecified order.

3.2 Communication and Synchronization in Linear Janus

In Linear Janus, both communication and synchronization happen in *channels*. Agents asynchronously send messages and post methods to channels. In a channel, if a message and a method match, a new agent is created. If a message (resp. method) does not match with any existing method (resp. message) in the channel, then the message (resp. method) stays in the channel until a matching method (resp. method) eventually appears, or until there is no agent left with access to that channel.

Communication (sends/posts) to a channel is asynchronous and the contents of a channel are unordered. Moreover, communication is many-to-many, i.e. multiple agents can share the same channel. The choices present a coherent picture of concurrent communications which is easy to understand and implement.

Logically, *sending* a message and *posting* a method in Linear Janus are similar to *tell* and *ask* operations in CCP. An important difference is that information in a channel is not necessarily monotonic. A message is consumed when it matches a method. A method is also consumed when it matches a message, unless the method is declared as pervasive.

The formal basis for the model described above is provided by a first-order intuitionistic fragment of linear logic, over the connectives ($\otimes$, &, $\multimap$, !). Parallel composition is modeled by multiplicative conjunctions ($\otimes$) and indeterminate choices by additive conjunctions (&); linear implications logically provide the vehicle for transforming a communication into a collection of activities; parameterization is provided by universal quantifications while hiding by existential quantifications. Finally, to complete the picture, the of course ("!") modal operator gives us the ability to install reusable resources.

3.3 KLJ: Linear Janus Kernel

We start with the class of items. An item can be one of the following:

1. A name, notated by alphanumeric strings that begin with a lower-case letter:

$$n ::= \langle identifier\_starting\_in\_a\_lower\_case\_letter \rangle$$

2. An i-parameter that can take on names as values. i-parameters are notated (like Prolog variables) by alphanumeric strings that begin with an upper-case letter:

$$\begin{array}{rccl} (Name) & p & ::= & \langle identifier\_starting\_in\_an\_upper\_case\_letter \rangle \\ (i-parameter) & i & ::= & n \mid p \end{array}$$

A term is a tuple $(i_1, \ldots, i_k)$ of $k \geq 0$ items, or a t-parameter which can take on tuples as values. The empty tuple ($k = 0$) is written as $()$. We do not lexically distinguish between i-parameters and t-parameters; both are notated by alphanumeric strings that start with an upper-case letter. But an identifier must be used consistently as either an i-parameter or a t-parameter in each scope and this restriction can be checked by a simple scan of the program.

$$\begin{array}{rccl} (Tuple) & u & ::= & (i_1, \ldots, i_n) \\ (t-parameter) & q & ::= & \langle identifier\_starting\_in\_an\_upper\_case\_letter \rangle \\ (Term) & t & ::= & u \mid q \end{array}$$

Note that tuples cannot be nested. The fact that tuples cannot be nested is not crucial for the semantics, but it facilitates the distinction between i-parameters and t-parameters: only i-parameters can occur inside a tuple. Such a distinction is made because messages and methods can only be posted on names, not on tuples. This is a decision motivated by simplicity of implementation, not logical necessity.

Curly braces ("$\{$" and "$\}$") are used for bracketing. Any expression e can be replaced by $\{e\}$ without changing its meaning.

By abusing the notation we also use the meta-syntactic variables n,p,i,u,q, and t to range over the corresponding syntactic class. The context will make it clear whether the class or an element of the class is intended.

Agents and methods are defined by:

$$
\begin{array}{llcll}
(Agents) & A & ::= & 1 & \text{— Nil} \\
 & & | & i\, t & \text{— Message send} \\
 & & | & i\, m & \text{— Nonce Method send} \\
 & & | & !\, i\, m & \text{— Reusable Method send} \\
 & & | & n\hat{}A & \text{— Local name} \\
 & & | & A * A & \text{— Parallel composition} \\
 & & & & \\
(Methods) & m & ::= & t \multimap A & \text{— Base Method} \\
 & & | & m \mathbin{\&} m & \text{— Composite Method}
\end{array}
$$

The method $t \multimap A$ is called a base method (with head t and body A); the method $m\&m$ is called a composite method. In an agent $i\ m$ (or ! $i\ m$), i is said to be the channel for the method. In addition, the following syntactic restrictions must be obeyed:

> For no agent $i\ m$ (or ! $i\ m$) does a parameter occur more than once in the head of a method in m. Further, if i is a parameter, it does not occur in the head of a method in m.

Informally, method $t \multimap A$ is fired when there is a message that matches t in the channel for the method. When method $t \multimap A$ is fired with message m, m is used to substitute t in A. For a composite method $m_1\&m_2$, only one of the components will be fired (i.e. there is a choice between m_1 and m_2).

The formal semantics of the KLJ will be given in section 3.4 and the informal interpretations of the agents are as follows:

- **1**: **1** (the nil agent) is an agent which vanishes immediately.
- **Message Agent**: A message agent, represented as $i\ t$, is the simplest *atomic* agent. Operationally, a message agent $i\ t$ sends the term t to channel i. If there is a method in i which matches with t, t is used to fire the method.
- **Method Agent**: A method agent, written as $i\ m$ or ! $i\ m$, is complementary to a message agent. Informally, the agent $i\ t \multimap A$ posts the method $t \multimap A$ to the channel i. If there is no matching message, the method *suspends* on i until a matching message arrives. When there is a message matching t with a substitution θ for the i- and t-parameters in t or when such a message arrives, the method is *fired*, yielding the agent A with θ applied to it.

In the agent $i\ m$, m is nonce and m vanishes from i when fired and is said to be consumed. In the agent $!\ i\ m$, m is reusable and is never consumed. That is to say, after firing, the method recreates itself.

For example, we can define a forwarding agent $x\ M \multimap y\ M$ that forwards a message in x to y. Since the method is a nonce method, it disappears after forwarding one message. The method can be made reusable by prefixing the agent "!". The reusable forwarding agent $!\ x\ M \multimap y\ M$ forwards all messages in x to y.

- Intuitively, $n\hat{}A$ is an agent that binds new channel to n and behaves like A. We say A is the scope of n. Since agents are recursively defined, scopes can also be nested.
- **Composite Agent**: A composite agent $A * B$ is an agent in which A and B execute in parallel.

3.4 Semantic Model of Linear Janus

This section explains the operational semantics of Linear Janus and the relation between the operational interpretation and the corresponding logical interpretation in linear logic.

3.4.1 Basic Definitions

Definition 3.4.1 The set of free parameters of a syntactic expression is defined inductively as follows:

$$
\begin{array}{ll}
fp(n) & = \emptyset \\
fp(p) & = \{p\} \\
fp((i_1, \ldots, i_n)) & = fp(i_1) \cup \ldots \cup fp(i_n) \\
fp(q) & = \{q\} \\
\\
fp(\mathbf{1}) & = \emptyset \\
fp(i\ t) & = fp(i) \cup fp(t) \\
fp(i\ t\ \multimap\ A) & = (fp(A) \cup fp(i)) \setminus fp(t) \\
fp(!m) & = fp(m) \\
fp(n\hat{}A) & = fp(A) \\
fp((A * B)) & = fp(A) \cup fp(B)
\end{array}
$$

Definition 3.4.2 The set of bound parameters of an agent A are the parameters that occur in A but are not free in A.

The occurrences of parameters in the head of a method are considered binding occurrences — they bind occurrences of the parameters in the body of the method.

Note the asymmetry in i and t in the definition of $fp(i\ t\ \multimap\ A)$. In $X\ (Y, Z, c)\ \multimap\ Y\ (c)$, X is a free parameter while Y and Z are not.

Definition 3.4.3 An agent A is said to be closed if $fp(A) = \emptyset$.

Definition 3.4.4 Similarly, the set of free names of a syntactic expression is defined by:

$$
\begin{array}{ll}
fn(n) & = \{n\} \\
fn(p) & = \emptyset \\
fn((i_1, \ldots, i_n)) & = fn(i_1) \cup \ldots \cup fn(i_n) \\
fn(q) & = \emptyset \\
 & \\
fn(\mathbf{1}) & = \emptyset \\
fn(i\ t) & = fn(i) \cup fn(t) \\
fn(i\ t\ \multimap\ A) & = fn(A) \cup fn(i) \cup fn(t) \\
fn(!m) & = fn(m) \\
fn(n\hat{}A) & = fn(A) \setminus \{n\} \\
fn((A * B)) & = fn(A) \cup fn(B)
\end{array}
$$

Definition 3.4.5 The set of bound names of an agent A is the set of names occurring in A less the names that are free in A.

A bound name n is introduced in agent A using the form $n\hat{}A$.

Note that a closed agent is an agent that does not have any free parameters. It is allowed to have free names.

For any syntactic expression e and parameters X and Y, the notion of X is free for Y in e is that X can be substituted for Y in e without becoming captured. The same is true for an expression e and names x and y.

3.4.2 Operational Semantics

We define a configuration, represented by Γ, to be a multiset of agents. We assume that agents are identified up to the renaming of bound parameters. The operational semantics defines the transition of a configuration of closed agents into another configuration of closed agents. The transition relation, $\longrightarrow$, is defined as the smallest relation closed under the following rules, where $\equiv$ stands for syntactic identity:

$$(1) \qquad \Gamma, \mathbf{1} \longrightarrow \Gamma$$

$$(2) \qquad \Gamma, A * B \longrightarrow \Gamma, A, B$$

$$(3) \qquad \Gamma, n\verb|^|A \longrightarrow \Gamma, A \quad (n\ not\ free\ in\ \Gamma)$$

$$(4) \qquad \frac{for\ some\ i, \theta, \quad t \equiv t_i\ \theta}{\Gamma, n\ t, n\ t_1 \multimap A_1\ \&\ldots\&\ t_k \multimap A_k \longrightarrow \Gamma, A_i\ \theta}$$

$$(5) \qquad \frac{for\ some\ i, \theta, \quad t \equiv t_i\theta}{\Gamma, n\ t, !n\ t_1 \multimap A_1\ \&\ldots\&\ t_k \multimap A_k \longrightarrow \Gamma, A_i\theta, !n\ t_1 \multimap A_1\ \&\ldots\&\ t_k \multimap A_k}$$

In (4) and (5), θ is a substitution on the parameters occurring in t_i. More precisely, θ is a substitution on all the parameters, because syntactic equality is used and the message cannot contain any parameters.

Intuitively, rule (1) says that the nil agent will disappear from the configuration; rule (2) says that multiset union is just a parallel composition; rule (3) allows us to discard unnecessary channels that are introduced; rules (4) and (5) state that a nonce method disappears from the configuration when fired but a reusable method is persistent.

Definition 3.4.6 A configuration Γ is said to be closed if for all agents A in Γ, $fp(A) = \emptyset$.

Note that if $\Gamma \longrightarrow \Gamma'$, and Γ is closed, then Γ' is closed.

3.4.3 Relation with Linear Logic

To see how the operational semantics defined in the last section relates to linear logic, we define a translation from agents to linear logic formulas. The translation $[\![\ldots]\!]$ consists of separate translations for each of the different syntactic categories. The categories n, p, i, u, q, t are translated into first-order linear logic terms. The category A is translated into linear logic formulas.

The linear logic formulas that result from the translation make use of a 2-place predicate letter written as ":". No special properties are assumed for this predicate letter.

$$
\begin{array}{rcll}
[\![n]\!] & = & & n \\
[\![p]\!] & = & & p \\
[\![(i1,\ldots,in)]\!] & = & & (i1,\ldots,in) \\
[\![q]\!] & = & & q \\
[\![\mathbf{1}]\!] & = & & \mathbf{1} \\
[\![i\ t]\!] & = & & [\![i]\!] : [\![t]\!] \\
[\![A * B]\!] & = & & [\![A]\!] \otimes [\![B]\!] \\
[\![n\hat{\ }A]\!] & = & & \exists n.[\![A]\!] \\
\\
[\![i \mid t_1 \multimap A_1 \,\&\ldots\&\, t_n \multimap A_n]\!] & = & & [\![i]\!] : [\![t_1]\!] \multimap [\![A_1]\!] \\
 & & \& & \ldots \\
 & & \& & [\![i]\!] : [\![t_n]\!] \multimap [\![A_n]\!] \\
\\
[\![!\ i \mid t_1 \multimap A_1 \,\&\ldots\&\, t_n \multimap A_n]\!] & = & ! & [\![i]\!] : [\![t_1]\!] \multimap [\![A_1]\!] \\
 & & \& & \ldots \\
 & & \& & [\![i]\!] : [\![t_n]\!] \multimap [\![A_n]\!]
\end{array}
$$

The above translation extends to multisets of agents element-wise, i.e.:

$$[\![A_1,\ldots,A_n]\!] = [\![A_1]\!],\ldots,[\![A_n]\!]$$

The linear intuitionistic proof rules that define the $\vdash$ relation between multisets of linear logic formulas and singletons are given in appendix A.

The basic connection between the operational semantics ($\longrightarrow$) and logical definition of $\vdash$ is as follows:

> Let Γ be a multiset of agents. Then $Gamma \longrightarrow \Gamma'$ implies that for any formula D with free names in $fn(\Gamma)$, we can derive a proof of $[\![\Gamma]\!] \vdash D$ from a proof of $[\![\Gamma']\!] \vdash D$.

For example, consider the following configuration Γ:

$$\Gamma \equiv x\hat{\ }y\hat{\ }z\hat{\ }\{x\ t * \{x\ M \multimap y\ M\} * \{x\ M \multimap z\ M\}\}$$

Starting with Γ, we can eventually reach a configuration Δ_1 or Δ_2 defined as follows:

$$\Delta_1 \equiv y\ t\ ,\ x\ M \multimap z\ M$$

or

$$\Delta_2 \equiv x\ M \multimap y\ M\ ,\ z\ t$$

Operationally, $x\ t$ sends a message t to x and the message will indeterminately activate either $x\ M \multimap y\ m$ leading to Δ_1, or $x\ M \multimap z\ M$ leading to Δ_2. We can formalize the reasoning by translating Γ into the corresponding linear formulas:

$$\exists x, y, z.(x : t \otimes \forall m.(x : m \multimap y : m) \otimes \forall m.(x : m \multimap z : m))$$

Then, a configuration Γ' is reachable from Γ if for any D with free names in Γ, we can derive $\Gamma \vdash D$ from a proof of $\Gamma' \vdash D$, i.e.:

$$\frac{\Gamma' \vdash D}{\Gamma \vdash D}$$

For instance, we can prove Δ_2 is a reachable configuration from Δ by the following proof:

$$\cfrac{\cfrac{\cfrac{\cfrac{\overline{x : t\ \vdash\ x : t} \qquad \overline{\forall m.x : m \multimap y : m, z : t\ \vdash\ D}}{x : t, \forall m.x : m \multimap y : m, x : t \multimap z : t \vdash D}}{x : t, \forall m.x : m \multimap y : m, \forall m.x : m \multimap z : m \vdash D}}{x : t, \forall m.x : m \multimap y : m \otimes \forall m.x : m \multimap z : m \vdash D}}{x : t \otimes \forall m.x : m \multimap y : m \otimes \forall m.x : m \multimap z : m \vdash D}$$

A similar proof can be constructed for Δ_2.

Unlike classical logic, there is no proof of $\Gamma \vdash D$ from a proof of $\Gamma_1, \Gamma_2 \vdash D$. In other words, Δ_1 and Δ_2 cannot both be reachable since there is only one message sent to x and the message is consumed by the method that is fired.

In Δ_1 or Δ_2, there is a *residue* method in x after one of the two methods consumes the only message sent to x. Since Γ is closed, this is inconsequential albeit inelegant. A cleaner configuration can be set up as follows:

$$\Gamma' \equiv x\hat{}y\hat{}z\hat{}\{x\ t * \{x\ M \multimap y\ M\ \&\ x\ M \multimap z\ M\}\}$$

The configuration Γ' will reduce to either $\Delta_1' \equiv y\ t$, or $\Delta_2' \equiv z\ t$.

3.5 Discussion

3.5.1 Concurrency and Indeterminacy

Communication and synchronization have always been regarded as the two faces of same coin and both are captured by using channels. In Linear Janus, the only important concept that one need to understand is the symmetry between senders and receivers of a channel. In our traditional understanding of concurrent systems, we always conceive of a set of processes which are synchronously or asynchronously activated by incoming messages. Even in object-based or actor-based systems, the common notion is that there is a communication port where an agent receives its messages which then activates further activities. Whereas a method imposition $x\ t \multimap A$ may be regarded as an agent which reads a message from x and behaves like A, we choose to interpret it as an agent that simply posts the method to x. The action of removing a message or firing a method all happens within a channel. There is no explicit notion of senders and receivers. An immediate consequence of such bilinear nature of channels is that communication is many-to-many, as opposed to one-to-one in process-based models or many-to-one in object-based and actor-based models. Concurrency arises because autonomous activities and interactions take place simultaneously. Indeterminacy arises because communication is asynchronous (i.e. arrival indeterminacy) and because channels can be accessed in a non-deterministic fashion (i.e. indeterminate choice).

3.5.2 Order and Fairness

The choices for asynchronous communication and unordered buffers are motivated by the fact that they closely reflect existing and emerging concurrent architectures and can be implemented efficiently. Moreover, synchronous and/or ordered communication can be readily implemented in our system, but not vice versa. However, the asynchronous unordered natures of communication post a question about fairness in our system.

Consider a server which receives non-deterministic requests from several clients. Can we guarantee that the server will eventually service each request? What happens if there is a malicious client sending an infinite sequence of requests to the server? If the server always services the shortest request first, for example, there is a possibility that some requests will never be answered. A simple solution is make the server enforce a first come first serve policy. Now, if a client cannot send infinite number of requests within a finite amount of time, fairness will be guaranteed. However, this requires a total ordering of incoming requests.

While the notion of ordered arrival guarantees fairness, the price is dear. Total ordering means that messages have to be queued and each message queue can be accessed only sequentially. In actors, messages in a mailbox are recorded in their arrival order and message handling is implicitly serialized since an actor can only deal with one message from the mailbox and has to spawn a replacement to take care of the rest of the message. This is probably the logical choice for actors because mailboxes are inseparable from their actors and communication is modeled as many-to-one.

On the other hand, communication is many-to-many in Linear Janus and channels do not belong to any particular agent. A channel may have multiple askers and tellers and they can access the channel in non-deterministic ways. It is pointless to enforce a total ordering of messages when there is so much uncertainty in the way they may be *observed*. The ordering in asynchronous systems is weak since it only requires that causality be respected, which is done in Linear Janus. With unordered messages and methods, we allow simultaneous accesses to channels. The price we pay is a lack of fairness guarantee, a shortcoming that we hope to mend when the semantics of concurrency is better understood.

Chapter 4

Computations in Linear Janus

This chapter discusses a number of examples to illustrate the computational properties of Linear Janus. The first part of this chapter introduces a number of syntactic shorthands to simplify writing. The second part presents a series of examples to illustrate techniques for composing concurrent or parallel computations.

4.1 Syntactic Abbreviations

While the syntax given in section 3.3 is sufficient for writing applications, it is rather verbose. For example, the agent for computing the factorial number would be rather lengthy:

```
! fact(N,R) ⊸
     one ^ { one(1) *
            tmp ^ { leq(N,one,tmp) *
                   tmp (true) ⊸ r(1) &
                       (false) ⊸ sR ^ { sub(N,one,sR) *
                                        fR ^ { fact(sR,fR) *
                                              times(fR,N,R) } } } }
```

To make our programs more concise, we introduce a number of syntactic abbreviations in this section. The abbreviations are expanded from inside out and from left to right.

1. For clarity purposes, multiple channels can be introduced together. That is, the agent $x\hat{}y\hat{}A$ is same as the agent $\{x, y\}\hat{}A$ if $x \neq y$. Furthermore, the agent $A * x\hat{}B$ is the same as $x\hat{}\{A * B\}$ if x is not in $fn(A)$.

   Given the convention, `fact` can be written as follows:

```
! fact(N,R) -o
     { one,tmp } ^ { one(1) *
                     leq(N,one,tmp) *
                     tmp (true) -o r(1) &
                         (false) -o { sR,fR } ^ { sub(N,one,sR) *
                                                   fact(sR,fR) *
                                                   times(fR,N,R) } }
```

2. We let an agent of the following form, where $k \geq 1$ and $ini \geq 1$ for each i:

```
x | p11,...p1n1 -o A1
  & p21,...p2n2 -o A2
  & ...
  & pk1,...pknk -o Ak
```

   represent:

```
x (p11,...p1n1) -o A1 &
  (p21,...p2n2) -o A2 &
  ...
  (pk1,...pknk) -o Ak
```

   The abbreviated form is called the case agent. With case agent, `fact` appears as follows:

```
! fact(N,R) -o
     { one, tmp } ^ { one(1) *
                      leq(N, one, tmp) *
                      tmp | true -o r(1)
                          & false -o { sR, fR } ^ { sub(N, one, sR) *
                                                    fact(sR, fR) *
                                                    times(fR, N, R) } }
```

3. We extend the class of items with $i ::= i\ u$. Let A be the syntactically smallest agent containing a term $i\ (i_1, \ldots, i_n)$. Then A is taken as a shorthand for the agent:

$$\mathtt{x\^\{\ i(i_1, \ldots, i_n, x)\ \ *\ \ a[\,x\ /i\ \ (i_1, \ldots, i_n)\,]\ \}}$$

where x is some name not in $fn(A)$. The intended meaning is that $i\ (i_1, \ldots, i_n)$ designates a channel on which the "result" of $i\ (i_1, \ldots, i_n)$ will be returned.

For example, consider the following abbreviations and their expansions, where $A \sim B$ means A is syntactically identical to B after the expansion of abbreviations in A:

```
(1)   f(c, a  b)  ~  x ^ { a(b, x)  *  f(c, x) }

(2)   f(c, a  b  d)  ~  x ^ { a(b, x)  *  y ^ { x(d, y)  *  f(c, y) } }

(3)   a(b){ 1 -o x3 & 2 -o x4 }  ~  y ^ { a(b, y)  *  y{ 1 -o x 3 & 2 -o x 4 } }
```

Note that such terms can be nested, i.e. the agent $x(y, z)(c, d)(P, Q) \multimap A$, with full paranthesization, stands for:

```
{ { x(y, z) }(c, d) }(P, Q) -o A
```

which expands to:

```
a₁^{ x(y,z,a₁)  *  { a₁(c,d) }(P,Q) -o A }
```

and further expands to the following:

```
a₁^{ x(y,z,a₁)  *  a₂^{ a₁(c,d,a₂)  *  a₂(P,Q) -o A } }
```

Intuitively $x(y, z)(c, d)(P, Q) \multimap A$ reads: the result of sending (y, z) to x is sent (c, d), and the result of that is tested with (P, Q), and reduced to A if the test succeeds.

With this abbreviation, we can abbreviate `fact` as follows:

```
! fact(N,R) -o
     one ^ { one(1) *
            leq(N,one) | true -o r(1)
                       & false -o times(fact(sub(N,one)),N,R) }
```

4. In addition, we further extend the class of items with $i ::= \hat{}t$. Let A be the syntactically smallest agent containing a term $\hat{}t$. Then agent A is taken as shorthand for the agent:

```
x^{ x t  *  A[ x /^t ] }
```

where x is a name not in $fn(A)$.

Intuitively, a subterm $\hat{}t$ stands for some channel on which the message t has been sent.

In addition, i-parameters are distinguished from t-parameters by examining their binding occurrences. If the parameter occurs inside a tuple, it is an i-parameter; if the parameter occurs as the head of a method, it is a t-parameter.

We allow i's to occur in places that t's are expected: in such a case an i is taken to stand for the 1-tuple (i). For example, consider the following abbreviations and their expansions:

```
(1)   f(c, ^ 1)  ~  x ^ (x 1 * f(c,x))
```

```
(2)   ^ 1{ { 1 ⊸ x ok } & { 2 ⊸ y ok } }  ~  y ^ { y 1 * y{ 1 ⊸ x ok & 2 ⊸ x ok } }
```

With implicit channels, `fact` can be abbreviated as follows:

```
! fact(N,R) ⊸
     leq(N, ^ 1) | true ⊸ r(1)
                 & false ⊸ times(fact(sub(N,^1)),N,R)
```

5. Finally, we extend the class of tuples with $u ::= \$i$. Let A be the syntactically smallest agent containing a term $\$i$. Then agent A is taken as shorthand for the agent:

```
x^{ x M ⊸ A[ M / $i ] }
```

where x is a name not in $fn(A)$ and M is a parameter not in $fp(A)$. For example, the forwarding agent $x\ M \multimap y\ M$ can be written more concisely as $y\ \$x$.

4.2 Concurrent Computations

Recursive data structures, such as lists, are very useful and common in functional and logic programming. However, lists are not very suitable in concurrent setting because they are built and accessed sequentially. Instead, a stream-like data is preferred because such a data structure allows us to deal with infinite and incomplete data. In Linear Janus, streams can be defined using 2-tuples where the first element of the 2-tuple is a term and the second element is another channel containing another 2-tuple or an empty tuple (signaling the end of the stream). For instance, a stream containing 1, 2, 3 can be written (using implicit channel abbreviation) as:

```
^ (1, ^ (2, ^ (3, ^ ())))
```

Similar to the function append for lists, we can define an agent `append` to concatenate two streams as follows:

```
! append(X,Y,Z) ⊸
     X | () ⊸ Z $Y
       & (H,T) ⊸ Z(H,append(T,Y))
```

Operationally, `append` functions as follows. First `X`, which is a channel presumably with only one message, is tested. If the content of `X` is the empty tuple `()`, the *content* of `Y` (i.e. `$Y`) is output to `Z`. If the message is a 2-tuple (with elements `H` and `T`), the output to `Z` is a 2-tuple whose first element is `H` and whose second element is a stream produced by recursively appending `T` and `Y`.

4.2.1 Dataflow Computation

The first example is an agent `or` which receives inputs from channels `a` and `b` and computes the logical disjunction of the two and produces the result on channel `c`. In Scheme, a similar function can be written as:

```
(define (or a b) (if a t b))
```

In a concurrent setting, one can imagine the scenario where some agents produce results on channels `a` and `b` while another agent waits for the result on `c`. If `a` and `b` are produced by agents that are arbitrarily fast or slow, how do we know which input should be tested first to achieve the best performance? The optimal order is to test whichever arrives first. To do this, the input messages from `a` and `b` are directed to a common channel `d` and the messages on `d` are then tested. Effectively, the channels `a` and `b` are merged into `d`. The `or` agent can be defined as below:

```
! or(A,B,C) ⊸
     d ^ { d $A * d $B * d | true ⊸ C(true) & false ⊸ C $d }
```

Operationally, `or` creates a local channel `d` and forwards the messages from `a` and `b` to `d`. To output the result, a message from `d` is examined. If the message is *true*, *true* will be output to `c` and terminate, otherwise, the next message of `d` is forwarded to `c`. During the process, some methods may suspend because the messages are not available, but, regardless of the arrival order of messages, a dataflow network of agents is set up and is activated by the arriving tokens.

4.2.2 Producer and Consumer

The example in the last section is a simplified case of producer-consumer computation. The general situation is that an agent (the producer) outputs some values which are used by another agent (the consumer). Normally, the producer is not in synchrony with the consumer, i.e. they produce and consume values at different rates. In synchronous systems, the faster agent has to wait for the slower one. We can eliminate such inefficiency by installing a buffer between the producer and the consumer, so that values can be produced and consumed out of sync. However, if the buffer is finite, the producer may overflow the buffer by generating values too fast. To avoid this problem, we develop a protocol to model a bounded buffer communication. In a bounded buffer communication, the sender blocks unless there is an empty slot in the receiver's buffer. In Linear Janus, we can implement a buffer by an open stream, i.e. a stream where new elements are being added to its end. For instance, we can install a stream L, e.g. `^(`p_1`,^(`p_2`, ... ^(`p_n`, more)))`, between the producer and the consumer. To proceed, the producer produces a value to the first channel of L and recurs with the second channel of L (which should be a stream). For the consumer, it waits for a value from the first channel of L and then appends a new pair to the end of L and recurs with the second element of L. A one-slot buffered communication can be implemented as follows, where `produce` and `consume` are the agents that do the real work:

$$
\begin{array}{l}
\mathtt{p}\ \hat{}\ \{\ \mathtt{producer(p, consumer(p))}\ *\\
\quad !\ \mathtt{producer(P, More)} \multimap \{\ \mathtt{produce(P)}\ *\ \mathtt{producer(\ \$More)}\ \}\ *\\
\quad !\ \mathtt{consumer(P, More)} \multimap \mathtt{p}\ \hat{}\ \{\ \mathtt{consume(P)}\ *\ \mathtt{More(p, consumer(p))}\ \}
\end{array}
$$

Besides the single-producer-single-consumer scenario, another common situation is one in which multiple agents are working cooperatively or competitively on the same task. As mentioned before, if a channel contains a reusable method M, we can safely imagine that there are unboundedly many copies of M. If we interpret each message as representing a pending task and each method as a worker, we have infinitely many worker. Whereas in sequential setting we may run into a problem with buffer overflows, in a concurrent setting we may run into a problem with too many active agents. Instead of bounding the buffer size, the equivalent situation is to limit the number of active agents. This can be done easily given the ability to pass channels. Two channels, `producer` and `consumer`, are created. Then, n copies of `consumer` are sent to `producer`, where n is the limit of active producers and consumers. A reusable method is installed on `producer` which takes a copy of `consumer` and outputs a result to it. Similarly, a reusable method is installed on `consumer` which when activated by a message from the producer sends a copy of `consumer` to `producer` to maintain the number of available consumers. For example, the following code implements a case where the maximum number of active producers and consumers is limited to two and `produce` and `consume` are again the agents who do the real work:

```
{ producer, consumer } ^ { producer(consumer) *
                           producer(consumer) *
                           ! producer(P) -o produce(P) *
                           ! consumer(P) -o { consume(P) * producer(consumer) } }
```

If multiple consumers and producers are working on the same task, it is likely that some of them have to share a set of common resources. In the next section we show how channels subsume the functionality of mutexes, semaphores, monitors or critical regions and the likes.

4.2.3 Assignments and Side-effects

Assignments and side-effects are commonly used in imperative programming languages. However, they are difficult to reason in formal models of semantics, program verification and program transformations. In purely functional programming, the problems associated with assignments and side-effects are outlawed. While simplifying the semantics, such approach also limits the kind of applications that can be supported.

In Linear Janus, using a channel as a storage unit produce the appearance of assignment and side-effects. Sending a message can be seen as an assignment, except that a channel contains a *multiset* of values instead of just one value. Similarly, removing a message from a shared channel and subsequently sending it another message produces side-effects affecting other methods in the shared channel. For example, consider the `counter_creator` example defined below, where `add` and `sub` are agents that perform addition and subtraction:

```
! counter_creator(C) -o
     count ^ { count(0) *
              ! C | (inc,R) -o add(count,^1) | N -o { count(N) * R(N) }
                  & (dec,R) -o sub(count,^1) | N -o { count(N) * R(N) }
                  & (val,R) -o count         | N -o { count(N) * R(N) } }
```

Operationally, `counter_creator` is activated with a channel (bound to `C`); it creates a new channel `count` with initial value 0 and installs a method on `C` which responds to requests for incrementing, decrementing or reading the value of the count.

A counter `C` may simultaneously receive multiple requests to increment, decrement, or reading its count. However, the access to `count` is exclusive because when one method takes the content of `count`, the other accesses are blocked until the content is restored. Even though `C` may simultaneously service `inc`, `dec` and `val`, the results returned are guaranteed to be serializable.

4.2.4 Passing Agents with Channels

In functional languages, the ability to pass functions as arguments is of great power. In Linear Janus only channels can be transmitted in communications. However, we can still obtain the expressiveness of a higher-order system by passing channels.

For example, a `map` agent can be defined to *apply* an function to elements of a stream:

```
! map(F,L,R) ⊸
     L | () ⊸ R()
       & (H,T) ⊸ R(F(H),map(F,T))
```

For example, we can define:

```
! inc(N,R) ⊸ add(N,^1,R)
```

And, we can invoke `map` as follows to increment each element of a stream to obtain a new stream `r`:

```
r ^ { map(inc,^(1,^(2,^())),r) }
```

Agents can also be *composed* as follows:

```
! compose(F,G,FG) ⊸ ! FG(M,R) ⊸ R(F(G(M)))
```

To produce an agent which increments a number by 3, we can write:

```
inc3 ^ { compose(compose(inc, inc), inc, inc3)  *  ... }
```

4.2.5 Divide and Conquer

Another common idiom in concurrent computing is to spawn out computations in a tree-like (or more generally, graph-like) fashion and then collect result from each node by progressive merging.

Consider the quicksort algorithm for a stream:

QuickSort:

1. (End Test) If the input stream is empty, close the output stream and terminate.
2. (Divide) Choose the first element as the *pivot*. Partition the input stream into two streams, S and L, such that elements in S are smaller than or equal to the pivot, and elements in L are strictly greater than the pivot.
3. (Conquer) Recursively call quicksort with S and L to produce S' and L'.
4. (Combine) Output the concatenation of S' and L'.

The parallel quicksort in Linear Janus is implemented as follows:

```
! qsort(Ls, R) -o
     Ls | () -o R()
        & (H, T) -o { s, l } ^ { split(^(H, T), H, s, l)  *  append(qsort(s), qsort(l), R) }
```

where `split` is defined as follows:

```
! split(IN, P, S, L) -o
     IN | () -o { S()  *  L() }
        & (H, T) -o le(H, P) | true -o split(T, P, S(H), L)
                             & false -o split(T, P, S, L(H))
```

In functional programming language, the parallelism in the quicksort algorithm arises from the parallel invocations of the recursive calls on the sub-lists. In Linear Janus there is an additional level of parallelism due to *pipelined* executions. Since the tail of a stream is another channel, the recursive calls to `qsort` with `S` and `L` need not wait for the completion of *split*. Similarly, `append` can starting execution without waiting for `qsort` to finish. We will further investigate pipelined execution in the next section.

4.2.6 Streaming and Pipelining

In sequential settings, only one thread of computation is active at a time. Typically, a process runs to completion and then passes its result onto the next process. In concurrent computing, however, multiple agents can be active simultaneously and unnecessary serialization of agents will result in inefficiency. If a producer computes a large result (e.g. an infinite list), it would be inefficient (if not impossible) to idle the consumer until the producer finishes.

To exploit concurrency, the producer and consumer can be arranged such that the consumer can start before the producer finishes. The producer can break the large result into smaller blocks as partial results, and as soon as a partial result is ready, the producer passes it to the consumer along with a *continuation* channel where the producer will output further results.

As a concrete example, suppose we want to find all the prime numbers smaller than n, using the *sieve of Eratosthenes* whose operations are as follows:

Sieve of Eratosthenes:

1. Create a stream L of integer from 2 to n
2. If L is empty, then terminate.
3. Take the first element a_1 of L and remove all multiples of a_1 from L to produce L'
4. Repeat step 2 with L'

The algorithm works by removing multiples of the first element (a prime) from the stream at each iteration and inductively, the first element of L at the beginning of each iteration is a prime. The algorithm as described above is sequential, but it can be pipelined. Instead of starting a new iteration when the previous finishes, we can start a new iteration as soon as elements are being output to

L', i.e. iteration $i+1$ works on the stream that is being produced by iteration i. This allows each iteration to pipeline its results to the next iteration and multiple iterations can proceed in parallel[1]. In Linear Janus, this pipelined sieve algorithm can be written as follows:

```
! sieve(N,R) -o
    filter ^ { ! filter(L,S,Re) -o
                  L | () -o Re()
                    & (H,T) -o
                      divisible(H,S) | true -o filter(T,S,Re)
                                     & false -o Re(H,filter(filter(T,S),H)) *
               R(2,filter(iota(3,N),2) }
```

To be precise, the nested invocation, `filter(filter(T, S),H)`, in `sieve` starts off a new pipeline as soon as a new prime number is found.

4.2.7 Delayed Evaluation

Can we use `sieve` defined in last section to find the i^{th} prime number efficiently? We can *if* we know a number that is just bigger than the i^{th} prime, which is not easy. Alternatively, we can start with an infinite list and stop when we find the desired prime.

```
! iota(N,R) -o iota(inc(N),R(N))
```

In Scheme, infinite lists (i.e. streams) can be create with the special form *delay* which is not evaluated until called with *force*[1]. In Linear Janus, *delay* and *force* can be implemented using the usual mechanisms for communication and synchronization. For example, `delayed_iota` defined below is an agent that produces a stream of integers starting from I:

```
! delayed_iota(N,R) -o
    R(N)(NR) -o delayed_iota(inc(N),NR)
```

[1]The improved algorithm can theoretically apply to streams of infinite size to find all primes.

This is consumed by delayed_cons defined as follows:

```
! delayed_cons(S,R) -o
     S | () -o R()
       & (H,T) -o T(R(H))
```

The construction works as follows. In delayed_iota, a 2-tuple is returned where the first element is the number N and the second element is a channel more which, when receives a channel NR (supplied by delayed_cons), will recursively invoke delayed_iota with $N + 1$ and NR.

The above definition of delayed_cons has a problem: delayed_cons cannot be passed the same stream twice, since the first invocation has consumed the method in more already. In Scheme, the implementation of *force* has to make sure a delayed expression is evaluated only once because the delayed expression may induce side-effects. Similarly, we can amend delayed_iota as follows:

```
! delayed_iota(N,R) -o
     more ^ { R(N,more) *
             more(NR) -o { v,w } ^ { inc(N,v),
                                     delay_iota(v,w) *
                                     more(NR) *
                                     ! more(P) -o v | N -o { P(N,w) * v(N) } } }
```

In essence, a nonce method is defined on more which, when invoked the first time, forces the next element to be produced at v and installs a reusable method on itself. The new method handles messages by reading from v. The agent more(NR) serves to invoke the new method to read the content of v to NR.

4.2.8 Deadlock and Starvation

As demonstrated in the counter_creator example, a channel (count) can be shared among multiple agents. The protocol given there is sufficient for coordinating the access to a single shared resource. However, if agents depend on more than one shared resource, deadlocks may occur.

Consider the classic example of dining philosophers where five philosophers share a common round table. When a philosopher gets hungry, he tries to eat by picking up a pair of chop-sticks, one from his right and one from his left. Unfortunately, each philosopher share the left chop-stick with his left neighbor and the right chop-stick with his right neighbor. If more than one philosopher get hungry at the same time, then some philosopher may not be able to eat because his left or right or both chop-sticks are being used by his neighbors. In the worst scenario, all philosophers may simultaneously feel hungry and pick up their right chop-sticks. But, no one can proceed to eat since there is no more chop-stick on the table. A deadlock occurs.

There are heuristics to prevent deadlocks. The philosophers can pick up both chop-sticks at the same time, or they can flip a coin to decide which chop-stick to pick up first, or a waiter can be hired to watch over the chop-sticks. The first two solutions are not elegant and the last alternative creates a bottleneck.

A simple solution is to make the philosophers talk to each other when someone is starved. When a philosopher finds himself hungry but does not have both chop-sticks, he tells his left neighbor. If a philosopher P is eating and his right neighbor P_r starts talking to him, then P asks P_r to wait. Otherwise, P tells his left neighbor P_l that P_r is waiting for him to finish. When a philosopher receives a message telling him that he is waiting for himself to finish, he knows that there is a deadlock and he should put down his chop-stick and try later. The behavior of our intelligent philosophers can be represented as below:

```
! philosopher_creator(Me, Neighbors, Left, Right) -o state ^ {
        state(contemplating) *

        ! Me | getting_hungry -o
                 Right(free) -o
                    Left | free -o state(_) -o { state(eating) * Left(busy) }
                         & busy -o state(_) -o { state(waiting) *
                                                 Left(busy) *
                                                 Neighbors(hurry_up, Me) }

              & getting_full -o
                 state(_) -o { state(contemplating) *
                               Left(_) -o Left(free) *
                               Right(_) -o Right(free) }

              & (hurry_up, Who) -o
                 equal(Who, me)
                    | true -o state(_) -o { state(thinking) *
                                            Right(_) -o Right(free) *
```

```
                                    Me(getting_hungry) }
              & false ⊸ state | eating ⊸ { state(eating) * Who(wait) }
                               & waiting ⊸ { state(waiting) *
                                             Neighbors(hurry_up, Who) }

     & wait ⊸ Neighbors(hurry_up, Me) }
```

Our intelligent philosophers do not guarantee that there is no deadlocks. The best a philosopher can do is to realize there is a deadlock, give up his chop-stick, and try later and with luck he will succeed in obtain a pair of chopsticks eventually.

4.3 Summary

This chapter showed some examples of common programming idioms in Linear Janus. Although the language constructs in Linear Janus are simple, complex computation can be developed with remarkable flexibility. We showed how channels can deal with merging, assignments, side-effects, many-to-many communication and synchronization. In addition, by passing channels we can gain the power of a higher-order system, and by defining streams in terms of channels, we can exploit a finer level of parallelism via pipelined executions. Finally, by using asynchronous communications, deadlocks can be detected without undue communication and overhead.

Chapter 5

Related Work

Over the past three decades, various models of concurrency and indeterminacy have developed and evolved. Some of the more significant examples are Net Theory[46], CSP[10], ACP[8], CCS[36] and π-calculus[37, 38], actors[24, 4, 12] and CCP[50]. There are also numerous languages designed to address issues in concurrent computing. The rest of this chapter compares and contrasts our work with other formalisms and languages.

5.1 Petri Nets

Net Theory[46], developed by Petri in the early 1960's, was the earliest mathematical model of concurrency. Informally, a net consists of a set of *places* and a set of *transitions*. Each transition is a relation between two multisets of places called the *preset* and the *postset*. A state of the net is specified by a *marking*, which is a multiset of places.

Pictorially, a net is a bipartite weighted directed graph. In the usual representation, a circular node represents a place and a rectangular node represents a transition where incoming and outgoing arrows specify the preset and postset. Dots represent tokens and each place in a marking has a token. Since a place may have multiple occurrences in the marking, it may have multiple tokens, one for each of its occurrence in the marking. When a transition is *fired*, tokens are consumed from the incoming arrows and produced on the outgoing arrows. Depending on the initial markings and dependence relations, transition firings can either be sequential, indeterminate or concurrent. An example of a net is given in figure 5.1.

It is easy to encode a Petri net in linear logic. Each token at a place a may be encoded as an atomic formula a and transitions in the net may be translated to reusable linear implications. For example,

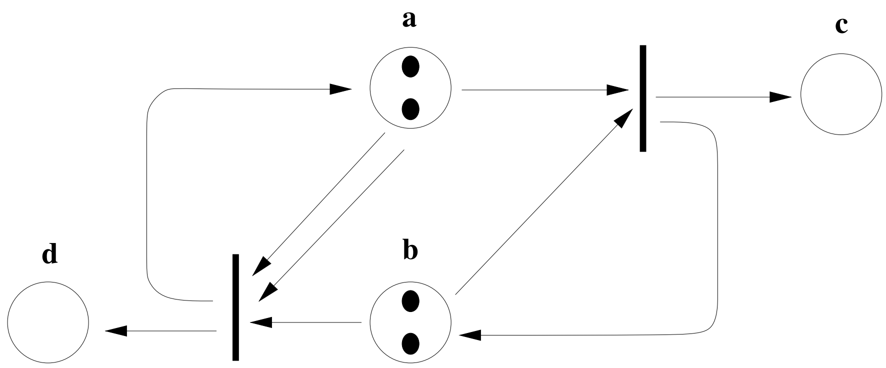

Figure 5.1: Petri net example.

the net presented in figure 5.1 can be encoded with a, a, b, b, $!((a{\otimes}b) \multimap (b{\otimes}c))$, $!((a{\otimes}a{\otimes}b) \multimap d)$. The firing of a transition is modeled as a discharge of the corresponding implication. *Reachability* in a net can therefore be encoded as the *derivability* of a sequent in linear logic. In the above example, $\{b, d\}$ is a reachable configuration from $\{a, a, b, b\}$ if and only if the following sequent can be proved:

$$a, a, b, b, !((a{\otimes}b) \multimap (b{\otimes}c)), !((a{\otimes}a{\otimes}b) \multimap d) \vdash b{\otimes}d$$

Indeterminacy in transition firings is readily reflected in the corresponding encoding in linear logic. Instead of $\{b, d\}$, the net may settle in $\{b, b, c, c\}$, which corresponds to the provable sequent:

$$a, a, b, b, !((a{\otimes}b) \multimap (b{\otimes}c), !((a{\otimes}a{\otimes}b) \multimap d) \vdash b{\otimes}b{\otimes}c{\otimes}c$$

A first-order Petri net can be encoded in Linear Janus using the scheme mentioned above. However, it would be incorrect to think that Linear Janus is similar to Petri Nets. Linear Janus provides substantially more power and expressiveness. In a net, tokens represent the *presence* of a condition, whereas in Linear Janus names represent channels which encompass more information than tokens. Channels in Linear Janus are sub-systems which can be transmitted in communications. The dependence relations and connectivities are static in a petri net but dynamic and reconfigurable in Linear Janus. Therefore, first-order Net Theory is better regarded as a sub-calculus of Linear Janus. Nonetheless, it would be fruitful to study properties of this sub-calculus more carefully since Net Theory has accumulated a rich body of results over the past three decades. Techniques have been developed to analyze properties like safeness and liveness of a net. It would be interesting to see how the experience can be carried over to Linear Janus.

5.2 π-calculus

Another formal approach to concurrency is through process algebra, as in ACP[8], CCS[36] and π-calculus[37, 38]. In this section, we discuss the connection between Linear Janus and π-calculus. Although Linear Janus started from the concurrent interpretation of linear logic while π-calculus was motivated by an algebraic reading of concurrency, the underlying principles between the two are remarkably similar.

In π-calculus, the syntax of agents are defined as follows[37], where x, y, z range over an infinite set of names $\mathcal{N}_\pi$, P over agents and A over agent identifiers:

$$
\begin{array}{lcll}
(agent)P & ::= & 0 & -inaction \\
 & | & \bar{y}x.P & -negative\ prefix \\
 & | & y(x).P & -positive\ prefix \\
 & | & \tau.P & -silent\ prefix \\
 & | & \Sigma_{i \in I} P_i, I\ is\ finite & -summation \\
 & | & \Pi_{i \in I} P_i, I\ is\ finite & -composition \\
 & | & (x)P & -restriction \\
 & | & [x = y]P & -match \\
 & | & A(y_1, ..., y_n) & -defined
\end{array}
$$

$\bar{y}x.P$ is an agent which outputs the name x at (output) port $\bar{y}$ and then behaves like P; $y(x).P$ is an agent which inputs an arbitrary name z at (input) port y and then behaves like $P[z/x]$ (simultaneous substitution); $\tau.P$ performs a silent action τ and then behaves like P.

0 is an agent that does nothing. $(x)P$ behaves like P except that actions at ports $\bar{x}$ and x are prohibited and an agent $[x = y]P$ behaves like P if the names x and y are identical, otherwise it behaves like **0**.

Agents are composed by either summation or composition. A summation $\Sigma_{i \in I} P_i$ behaves like P_i for some i, whereas $\Pi_{i \in I} P_i$ behaves like P_i for all i in parallel. Finally, $A(y_1, \ldots, y_n)$ is an agent with arity n defined previously by a defining equation $A(y_1, \ldots, y_n) \equiv P$.

A more detailed discussion can be found in[37].

As explained by Milner et al, the goal of π-calculus is to model systems in which components may be arbitrarily linked and communication may cause the linkages to be changed dynamically. It is the same kind of considerations that motivated the development of Linear Janus. Indeed the underlying principles of the two are so similar that a lot of the problems and solutions discovered in π-calculus can be directly carried over to Linear Janus. The encoding for lazy λ-calculus and combinatory logics are two of the examples that will be discussed in appendix B. In fact, there is a natural translation of π-calculus to Linear Janus: negative prefix can be translated to message send, positive prefix to method imposition, restriction to hiding, summation to indeterminate choice and composition to parallel composition. Match is not strictly necessary in π-calculus, and silent prefix and define are already embedded in indeterminate choice and method imposition in Linear Janus. Assuming $\mathcal{N}_\pi \subseteq \mathcal{N}$, the table below shows such a translation map $[\![.]\!]$ from π-calculus to Linear Janus:

$$\begin{aligned}
[\![\mathbf{0}]\!] &= \mathbf{1} \\
[\![\bar{y}x.0]\!] &= y(x) \\
[\![y(x).P]\!] &= y(X) \multimap [\![P]\!] \\
[\![(y)P]\!] &= y\wedge[\![P]\!] \\
[\![P+Q]\!] &= [\![P]\!] \,\&\, [\![Q]\!] \\
[\![P|Q]\!] &= [\![P]\!] * [\![Q]\!]
\end{aligned}$$

Although π-calculus and Linear Janus bear a lot of resemblances with each other, there are subtle differences between the two. The most important disparities arise from the different starting points of π-calculus and Linear Janus. The semantic model of π-calculus is developed via process algebra, whereas the semantics of Linear Janus is developed by adopting a concurrent interpretation of linear logic. As a result, processes equivalences in π-calculus are defined by bisimilarity, whereas in Linear Janus processes equivalences are defined a finer level by logical equivalences. Additionally, the seemingly mysterious notions of scope intrusions and extrusions of π-calculus are captured logically in Linear Janus by existential and universal quantifications. Using closures, Linear Janus also eliminates the need of the explicit restriction operator of π-calculus[1].

Both π-calculus and Linear Janus are first-order systems in that only *references* to processes are passed in communications. In section 6.1.3, we will discuss the ramifications of moving to a higher-order system.

[1]The scoping in Linear Janus is lexical, while dynamic scoping can be implemented in π-calculus using restriction and scope extrusion.

5.3 Actors

Linear Janus started from the actor model of computations, so naturally there is a very close relationship between the two and many similarities have been discussed in the last few chapters. However, the two differ in some fundamental aspects. First, communication in Linear Janus is many-to-many, instead of many-to-one in actors. Mailboxes in actors are implicitly part of the actors, while in Linear Janus, channels are symmetric links among agents. As shown in chapter 4, such symmetry allows a richer pattern of communication that are useful in real systems. Second, in Linear Janus, new agents can dynamically install methods on pre-existing channels (such as in `counter_creator`, which is impossible in actors)[2]. Such ability allows agents to dynamically modify the behavior of a system.

The treatments of recursion and state changes in the two are also different. A serialized actor, when invoked, needs to specify a replacement for itself to achieve recursion and passes on any state information to the replacement. Although the replacement can be concurrently computed with other events in a script, there is an notion of sequential access to messages due to arrival order. As mentioned in section 3.5.2, the mail system of actors serializes concurrent communications and creates a total ordering for messages in each mailbox. On the other hand, in Linear Janus, reusability gives rise to recursion and arbitrarily many copies of a reusable method can be *activated* at the same time by messages. Channels allow concurrent asks and tells and the only serialization is due to resource availability. Linear Janus agents are inherently more concurrent than actors.

Finally, an actor can only service one mailbox which is its only port for receiving communications. An immediate consequence is that actors are not directly compositional. A configuration of actors, in general, is not itself an actor since it can respond to messages arriving at different mailboxes (ports). The problem can be solved by introducing the concept of *receptionists* to handle interactions among configurations, but introducing a special actor interface also increases the complexity of actor semantics. On the other hand, a configuration of agents in Linear Janus is itself an agent. Configurations can be composed just like agents and can allow multiple interfaces to convey different *views* of the system.

[2]This opens the question about security, which will be addressed in the next chapter.

5.4 Chemical Abstract Machine

In this section, we informally discuss the Chemical Abstract Machine[9] and describe the close resemblance between Linear Janus and a chemical abstract machine, as a way to discover any complexity issues and possible implementation difficulties. As put forth in [9], constructing a chemical abstract machine (CHAM) is like standard programming; if a concept is difficult to realize in a cham, it is likely to be difficult to be implemented on a real machine.

Intuitively, CHAM is analogous to a chemical solution in which molecules float around freely and interact with each other according to a set of chemical equations. The elegance of a chemical abstract machine is that molecules do not explicitly interact with one another. Motions of molecules are governed by some system dynamics (Brownian motion, concentration gradient, for example) and reactions occur only when complementary molecules randomly run into each other.

To draw the analogy even closer, a chemical solution may be *heated* in order to break large molecules into their constituents called *ions*, or *cooled* so that molecules can combine to form compound molecules.

A solution may also contain *subsolutions* enclosed in *membranes*. Subsolutions can evolve independently and can be part of a molecule, i.e. it is itself a molecule. In order to allow the molecules inside a subsolution to react with those outside, membranes are porous. A single molecule may also be isolated within an *airlock*. In computational terms, subsolutions provide the abstraction layers and airlocks allow interfaces to be exported.

Using an example given in [9], we can construct a chemical abstract machine to find the prime numbers between 2 and n as follows. We take integers as our molecules and start with a solution containing all integers from 2 to n. We specify one reaction rule: an integer reacts with its multiples by destroying them. When the solution reaches equilibrium eventually, we will have prime numbers between 2 to n.

Channels, which are the basis of Linear Janus's computational model, are identical to what was called a subsolution in the above. The connection between concepts mentioned above and constructs in Linear Janus are quite obvious. The logical structural rules of Linear Janus can be easily translated into heating and cooling rules, while logical transitions of agents can be translated into reaction transformations. The implementation of Linear Janus on CHAM is not terribly interesting in itself. However, by comparing implementations of Linear Janus with those of similar calculi, such as TCCS[42], we can see that the implementation of Linear Janus can be simpler. The restriction operator in TCCS (and π-calculus) requires the airlock mechanism, which is not necessary for Linear Janus. Additionally, external sum in CCS was shown to be inappropriate for CHAM and thus raises the question about the feasibility of external sum in a real implementation. On the other hand, all the basic ideas in Linear Janus correspond to those of CHAM closely. This gives us more confidence to say that the our framework is feasible both as a theoretical model and a practical tool. We will discuss the real implementation in the second part of the thesis.

5.5 Linda

The computational model presented by Linda[3] centers around the notion of tuple spaces. A tuple space resembles a shared associated memory where processes can communicate by reading and writing messages represented by tuples. Writing is done by **in**(TS, T), which adds a tuple T to the tuple space pointed by TS; reading is done by **out**$(TS, \langle P, ?M\rangle)$, which instantiates M to M' if there is a tuple $T = \langle P', M\rangle$ in TS such that P matches P'. The matched tuple T in an **out** operation is removed from the tuple space during the process. A third primitive **rd** behaves like **out**, except that the matched tuple is not removed from the tuple space.

Conceptually, Linda is very similar to Linear Janus. First class tuples spaces[21] are like channels and Linear Janus may be viewed as a formalized version of *higher-order* Linda (such as in [18]). On the other hand, the two differ rather radically in the uniformity, modularity and scalability of their communication primitives. Linda is better thought of as a coordination language that adds a communication layer on top of some languages (C-Linda, S-Linda and M-Linda). In Linda, local communications (e.g. communications within the same program) are done by native primitives and only global communications (e.g. communications among programs) are supported by tuple spaces. However, in Linear Janus channels are the only primitive[3]. Additionally, each channel is an *active* component of computation bringing together messages and methods related for a specific task. A tuple space, on the other hand, is more like a (gigantic) message server that centrally handles messages from possibly disparate processes. Channels are extremely lightweight and mobile since each channel is a minimal entity capturing only the essential part of an autonomous task. Efficient Linda implementations have to resort to optimizations that depend on global knowledge of the systems, making the language unsuitable for open and distributed systems. As we shall see in the second part of the thesis, the feather-weight nature of channels allows an implementation which can scale from shared memory multiprocessors machines to distributed networks of computers.

[3]To be fair, the predominant goal of Linda is to provide interoperability in a multi-lingual environment besides concurrency, while in Linear Janus, we try to develop a uniform conceptual framework for understanding concurrency and indeterminacy. In the second part of the thesis, we will discuss issues related to interoperability.

5.6 ACL

Kobayashi and Yonezawa have investigated a language called ACL[22] simultaneously but independently of our work. The fragment of linear logic used in ACL is dual of that used in Linear Janus. In ACL, both messages and processes are represented as logic formulas. For example, in ACL $m \; por \; A$ is a process that sends a message m and then becomes A, while $m^{\perp} \otimes B$ is a process that receives a message m and then becomes B. A choice between two processes A and B is represented by $A \oplus B$. In ACL, counters can be expressed as follows (taken from an example in [22]):

$$
\begin{aligned}
Counter(Id, N) \circ\!- \; & (inc(Id)^{\perp} \otimes Counter(Id, N+1)) \\
& \oplus \exists Rid(read(Id, Rid)^{\perp} \otimes (val(Rid, N) \; por \; Counter(Id, N))) \\
& \oplus (reset(Id)^{\perp} \otimes Counter(Id, 0))
\end{aligned}
$$

The similarity between the above definition and our definition of `counter_creator` in section 4.2.3 is immediate.

However, Kobayashi and Yonezawa focus mainly on the abstract semantics of ACL and they do not seem to have investigated implementation issues. Indeed, their language seems difficult to implement. In ACL, a process which waits for messages $m_1, \ldots, m_n$ and becomes P is represented as:

$$m_1^{\perp} \otimes \ldots \otimes m_n^{\perp} \otimes P$$

which is logically equivalent to:

$$m_i^{\perp} \otimes m_1^{\perp} \otimes \ldots \otimes m_{i-1}^{\perp} \otimes m_{i+1}^{\perp} \otimes \ldots \otimes m_n^{\perp} \otimes P$$

A problem with the above equivalence is that deadlock may occur when a process indiscriminately consumes messages that are needed for other processes to make progress, for example in:

$$((x^{\perp} \otimes y^{\perp} \otimes z^{\perp} \otimes P_1) \oplus (x^{\perp} \otimes y^{\perp} \otimes P_2)) \; por \; (x \; por \; y \; por \; \mathbf{1})$$

In addition, while the above equivalence provides a mechanism useful for synchronization between multiple processes and for splitting large data into multiple messages, its implementation seems non-trivial. An important aspect of Linear Janus is that communication and synchronization only occur through channels, and one message at a time. This restriction allows an efficient implementation while still retaining the expressiveness to embed formalisms like λ-calculus, π-calculusand etc, as shown in the previous sections. However, neither this thesis or [22] develops a denotational semantics that will distinguish between the following two agents:

$$
\begin{aligned}
\mathtt{A}_1 \equiv \mathtt{z}(\mathtt{Q}) &\multimap \mathtt{x}(\mathtt{M}) \multimap \mathtt{A} \\
\& \ (\mathtt{Q}) &\multimap \mathtt{x}(\mathtt{M},\mathtt{N}) \multimap \mathtt{B}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{A}_2 \equiv \mathtt{z}(\mathtt{Q}) \multimap \{\ &\mathtt{x}(\mathtt{M}) \multimap \mathtt{A} \\
\& \ &(\mathtt{M},\mathtt{N}) \multimap \mathtt{B}\ \}
\end{aligned}
$$

Logically A_1 and A_2 are equivalent. However, if x only has 2-tuple message, A_1 may deadlock while A_2 will not. In order to distinguish A_1 and A_2, we have to take a step beyond linear logic. There is much work to be done to develop a useful denotational semantics for the language.

Chapter 6

Future Work

This chapter discusses a number of extensions that are being developed and directions that Linear Janus may evolve in the future. Section 6.1 first examines a number of theoretical issues, such as type systems, proof theoretical models and higher-orderness. Section 6.2 considers some pragmatic aspects of Linear Janus, such as security and reflectivity.

6.1 Theoretical Issues

6.1.1 Polymorphic Type System

One of the advantages of Linear Janus is its sound theoretical foundation, which allows us to devise powerful mathematical and logical tools to support software development. In well-developed high-level languages like ML[40], Miranda[58] and others, polymorphic type systems[35] have been proven to be invaluable in both the development and maintenance of software. Saraswat is developing a linear polymorphic type system for linear `cc` language by exploiting the proofs-as-types principle. Within the current formulation of Linear Janus, we are able to type check a program. However, the main challenge is to integrate concurrency and indeterminacy and to devise an efficient inference algorithm to deal with issues with dynamic process creations and method installations on pre-existing channels.

6.1.2 Proof Theory Model

In previous discussions, we have sometimes relied on our intuitions to specify properties of a program. However, one of the major motivations behind this research is to establish a solid foundation for clarifying semantic and operational meanings of a program, verifying correctness (such as termination) and detecting abnormalities (such as deadlock) in a rigorous framework. Currently, proving the interesting properties of a program is laborious, even with the help of linear logic. However, there are reasons to believe that many of the intuitions can be axiomatized to establish a more convenient proof theoretical model. Moreover, in the chapter 5, we have seen the connections between Linear Janus and a number of other well-developed formalisms. It will be rewarding to see what can be learned from the experiences gained in developing those formal systems and to find out if a different semantic model (e.g. algebraic theory) for Linear Janus can be developed to gain insights from outside logic.

6.1.3 Higher-Order Systems

In Linear Janus, as in π-calculus, agents are never passed directly in communication. Instead, references to agents are transmitted. As demonstrated in chapter 4, by passing channels only, we gain the effective power of a higher-order system without distracting ourselves with theoretical and practical issues associated with higher-order systems. Similar claims can be made of π-calculus and in [37] the rationales were summarized by the following passage:

> "First to pass R as a parameter to $Q \equiv y(z).Q'$ may result in replication of R, due to repeat occurrence of the formal parameters z within Q'; we do not wish the replication of agents with state to be a primitive of the π-calculus. Second, to pass R as a parameter gives Q access to the *whole* of R; we are concerned to model the case where a received name x provides only partial access to another agent. (For example, R may communicate with still other neighbors via names not known to Q.) Third, the transmission of access links is a very common phenomenon in computation, even in purely sequential computation, which has hitherto had no adequate theoretical basis; we must examine primitives which may provide such a basis in as lean a framework as possible."

Nevertheless, the development of higher-order versions of π-calculus and linear cc is a challenging area of research and proposals are emerging. For linear cc, Saraswat and Lincoln[52] are developing a higher-order framework by moving onto higher-order linear logic. In higher order systems, processes are first-class citizens which can be communicated. By allowing processes in messages, agents can acquire and manipulate processes with complete freedom. Such freedom is the power behind λ-calculus and there are reasons to believe a higher-order linear cc system will lead us to a true concurrent λ-calculus. But much research has to be done to achieve such end.

6.2 Pragmatic Issues: Security and Modularity

The current formulation of Linear Janus is as lean as we could possibly make it. There is a minimal set of ideas built into the language and control-structures and data-types are derived from a basic primitive, namely channels. However, while such cleanness reduces the complexity of the theoretical treatment, it also hinders the pragmatic uses of Linear Janus. In many ways, this is an intentional choice since we are more concerned with a vigorous framework and do not want extraneous details to obscure the big picture. There are certainly a lot of improvements that can make Linear Janus a much better programming language.

For example, an important issue in real computer systems is security. Security of a language has to depend on the underlying implementation of the operating system and communication protocol. However, assuming there is a secure underlying platform, it is illuminating to consider the possible implementation of a secure system on the language level. In [cp], a secure electronic banking system was implemented in Concurrent Prolog, using unification and unforgeable read only variables of Concurrent Prolog. In [27], a similar problem was solved in GHC, which only required unforgeable unique constants.

Our formulation of Linear Janus is incapable of implementing a secure system, since communication is open. An agent can both ask and tell a channel to which it has access. Access to a channel can be duplicated (for example when a universally quantified name appears more than once in the body of a method). It is easy to have malicious agents to eavesdrop on communications and disrupt the system. However, a slight extension will allow us to implement a secure system. The basic idea is to introduce, as in Janus, the notions of *ask right* and *tell right*. Having the tell right to a channel, an agent can only send messages to that channel, while an ask right on a channel only allows the agent to post methods on that channel.

On a more interesting note, a class/module system for Linear Janus can be implemented by using the same notions of ask rights and tell rights. Agents within a class/module have one or both ask and tell rights to channels defined in the class, while agents from outside the class can only have tell rights to those channels. In other words, an abstraction is set up such that only agents within the same class can affect each others' behavior.

Chapter 7

Implementation of Linear Janus

Computations in Linear Janus evolve through the interaction of messages and methods at channels. The usage of channels to encapsulate both communication and synchronization provides a simple conceptual model. The price for the simplicity is the difficulty in implementing the compiler and the runtime system efficiently. This is the focus of this and the next few chapters.

Our main objective in developing a prototype system is to understand issues in implementation and the computational behavior of agents in Linear Janus. As a result, the focus is on generating a functional and flexible platform for performing experiments both in the compiler and in the runtime system.

Linear Janus is designed as a language capable of extremely fine-grained concurrent computation. Each agent performs a minute amount of computation. The easiest and simplest implementation would represent each agent as a process. However, current generation of concurrent machines are geared towards coarse-grained computations with few communications. In the T3D, for example, each node is an Alpha processor where instruction granularity is of the order of thousands of instructions and creating new processes and switching among processes are of the order of many hundreds of instructions. In such an architecture, the implementation of agent as a process is infeasible. On the brighter side, there are a few fine-grained concurrent machines, such as the J-machine, to handle the task. In the J-machine, each node is a message driven processor (MDP) specially designed to support object-based and actor-based computations. The MDP provides a global name space on top of distributed memory and supports (in the hardware) mechanisms for fast process creations, context switchings and internode communications. In an architecture like the J-Machine, there is considerably more freedom in choosing the granularity and mobility of agents. The J-machine would have been a good platform for implementing Linear Janus, had it developed a robust programming system.

The current implementation of Linear Janus was developed for Sun SPARCStations. In many aspects, SPARCStations, being the conventional workstations, are not very interesting as a platform for implementing Linear Janus. Conventional workstations are designed to run C-like procedural languages and Unix-like operating systems. Performance depends heavily on strong locality, long instruction sequences and stack-based execution. The support for investigating a highly concurrent language is virtually non-existent on SPARCStations, or other conventional workstations for that matter. On the other hand, given that our main objective is to understand issues in concurrency, a workstation with a robust programming environment suffices.

The current implementation defines an abstract machine architecture and a set of abstract machine instructions which serves as the target of our compiler. An abstract machine has two components, the kernel and the communication interface. The kernel provides the environment where local agents interact. The communication interface is responsible for coordinating agents on different abstract machines to form a bigger system. Currently, each abstract machine is implemented as a normal Unix process and multiple abstract machines can reside on the same machine or on different machines on the network.

The next four chapters of the thesis are organized as follows. Chapter 8 describes the details of the abstract machine design. Chapter 9 discusses the optimizations in our prototype compiler. In chapter 10, the implementation of the abstract architecture is explained. Finally, some preliminary observations about the current implementation of Linear Janus are discussed in chapter 11.

Chapter 8

Abstract Machine

8.1 Overview

The main reason for defining an abstract machine is to hide the runtime implementation issues from the compiler. With an abstract view of a machine, the compiler has a stable target architecture, while the runtime implementation may change over time.

Our emphasis on the abstract machine design is simplicity, sometimes at the expense of efficiency. In addition, to avoid unnecessary assumptions on the implementation of the abstract machine, services which can vary in different implementations are not considered part of the abstract machine definition, such as process scheduling and memory management.

The abstract machine consists of two components – one is responsible for dealing with communication among abstract machines, while the other carries out the local computations and communications. Operations in the abstract machine are defined by a set of abstract machine instructions, similar to conventional instruction sets.

The following sections describe the operations of the abstract machine, as well as a subset of abstract instructions.

8.2 AM Kernel

8.2.1 Abstracted Execution Model

The kernel of an abstract machine provides an abstract view of the environment in which agents interact to perform computations:

Initially, the environment consists of a set of agents. When an agent sends a nonce message or posts a nonce method to a channel, the following happens at the channel. When a message is sent to a channel, the methods on the target channel are examined. If there is no matching method, the message is added to the multiset of messages. Otherwise the message is used to instantiate the pattern of the method and to create a new agent. Similar actions happen when a method is posted. The messages on the target channel are examined. If there is no matching message, the method is added to the multiset of methods. Otherwise the message is removed and is used to instantiate the pattern of the method to create a new agent.

In the case of reusable messages and reusable methods, similar aforementioned actions are carried out, except that a reusable item is always added to the corresponding multiset regardless of whether there is a matching message or method. Moreover, by definition, a reusable message or method is never removed even if it is used to create a new agent.

8.2.2 Abstracted Representation for Message, Method and Agent

The abstract machine's representations for messages, methods and agents are as follows:

- **Message.** A message is represented as a tuple consisting of a number of terms (the number is referred to as the arity of the message).
- `Method.` A method consists of a pattern and an agent (the body). The pattern is like a message, except that some terms may be universally quantified. To match a message with a pattern, the corresponding terms are compared. If both are unquantified variables, their equality is tested. Otherwise the message term is used to instantiate the quantified term. When a match is completed, a new agent is created from the instantiated body of the method.
- **Agent.** An agent consists of three parts:
  1. Code Pointer: The code pointer points to a sequence of abstract instructions which represents the agent.
  2. Message Pointer (AP): The message pointer points to the message that is used to created the agent. In the code sequence, AP[i] refers to the i^{th} term in the message.
  3. Closure Pointer (CP): Linear Janus is lexically-scoped. Hence an agent may contain free variables which are defined in a lexical context enclosing the agent. In the code sequence CP[i] refers to the i^{th} variable in the closure.

8.3 AM Kernel Instructions

The complete list of instructions is included in appendix C. In this section, only the instructions which pertain to our examples are discussed and other instructions are introduced as their rationale becomes more obvious.

Our abstract machine instructions are similar to RISC machine instructions, i.e. each operand is either an immediate, a register or a register index. Moreover, we assume that there is an infinite number of registers, named by A[i], where $i \geq 1$. For clarity, operands are given meaningful names in the following examples.

The abstract instructions are classified into two sets, one is visible to the compiler while the other is not visible to the compiler but is used to implement the instructions that are visible. Instructions which are visible to the compiler are marked with an * in the following.

8.3.1 Data manipulation

The following instructions are for object creation and manipulation:

- Alloc(n, r) allocates a new record of `n` words and returns the pointer of the record to `r`.
- Store(s, d)* stores `s` to the location pointed by `d`.
- Deref(s, d)* returns the value stored at `s` to `d`.
- Arity(p, n)* returns the arity of record pointed by `p` to `n`.

The following set of instructions, which can be defined by the instructions given above, provide means for creating channel, message, method, pattern and closure:

- NewChannel(r)* returns a channel to `r`.
- NewClosure(n, r)* returns a closure of arity `n` to `r`.
- NewPattern(n, r)* returns a pattern of arity `n` to `r`.
- NewMessage(n, r)* returns a message of arity `n` to `r`.

- NewRMethod(p, cl, b, r)* returns a reusable method with pattern `p`, closure `cl`, and agent `b` to `r`.
- NewNMethod(p, cl, b, r)* returns a nonce method with pattern `p`, closure `cl`, and agent `b` to `r`.

8.3.2 Communication

The following instructions are defined to manipulate a channel:

- Send(msg, c)* sends message `msg` to channel `c`.
- Post(mth, c)* posts method `mth` to channel `c`.

8.3.3 Tests

- IsEq(p1, p2, cc)* returns `TRUE` to `cc` if `p1` and `p2` are equal.

8.3.4 Control

The control instructions are defined as follows:

- Jeq(p, q, lbl)* jumps to label `lbl` if `p` equals `q`.
- Jcc(cc, lbl)* jumps to label `lbl` if `cc` is `TRUE`.
- Jmp(lbl)* jumps to label `lbl` unconditionally.
- Ret()* terminates the agent.

8.3.5 Primitives

The set of primitive agents which perform simple arithmetic operations are defined as follows (the arguments are all channels):

- Add(p1, p2, r)* returns to `r` the result of adding `p1` and `p2`.

- Sub(p1, p2, r)* returns to `r` the result of subtracting `p1` from `p2`.
- Mul(p1, p2, r)* returns to `r` the result of multiplying `p1` and `p2`.
- Div(p1, p2, r)* returns to `r` the quotient of dividing `p1` by `p2`.
- Mod(p1, p2, r)* returns to `r` the remainder of dividing `p1` by `p2`.
- Eq(p1, p2, r)* returns to `r` the result of equality test for `p1` and `p2`.
- Lt(p1, p2, r)* returns to `r` if `p1` is less than `p2`.
- Le(p1, p2, r)* returns to `r` if `p1` is less than or equal to `p2`.
- Gt(p1, p2, r)* returns to `r` if `p1` is greater than `p2`.
- Ge(p1, p2, r)* returns to `r` if `p1` is greater than or equal to `p2`.

8.3.6 Internal Instruction

Finally, the following set of instructions are not visible to the compiler but is used for implementing the instructions mentioned above:

- MatchMethod(msg, c, mth) returns to `mth` a method in c that matches `msg`.
- MatchMessage(mth, c, msg) returns to `msg` a message in c that matches `mth`.
- AddMessage(msg, c) adds message `msg` to channel `c`.
- RemMessage(msg, c) removes message `msg` from channel `c`.
- AddMethod(mth, c) adds method `mth` to channel `c`.
- RemMethod(mth, c) removes method `mth` from channel `c`.
- New(msg, mth) creates a new agent by instantiating method `mth` with message `msg`.
- IsChannel(p, cc) returns `TRUE` to `cc` if `p` is a channel, `False` otherwise.
- IsMessage(p, cc) returns `TRUE` to `cc` if `p` is a message, `False` otherwise.

- IsRMethod(p, cc) returns `TRUE` to `cc` if `p` is a nonce method, `False` otherwise.
- IsNMethod(p, cc) returns `TRUE` to `cc` if `p` is a nonce method, `False` otherwise.

Given these instructions, `Send` can be implemented by the following sequence of instructions:

```
Send:
   MatchMethod(msg, c, mth);  - get the list of methods of c

   Jeq(mth, NULL, L_insert);  - if there is no method, insert the message
   IsRMethod(mth, cc);
   Jt(cc, L_new);
   RemMethod(mth, c);         - remove the nonce method

L_new:
   New(msg, mth);             - create a new agent with msg and mth
   Ret();

L_insert:
   AddMessage(msg, c)         - insert the message
   Ret();
```

Example

To briefly illustrate the use of the abstract instructions, consider the following example, first described in section 4.2.3.

```
! counter_creator(C) ⊸
     count ^ { count(0) *
              ! C | (inc,R) ⊸ add(count,^1) | N ⊸ { count(N) * R(N) }
                  & (dec,R) ⊸ sub(count,^1) | N ⊸ { count(N) * R(N) }
                  & (val,R) ⊸ count | N ⊸ { count(N) * R(N) } }
```

which can be represented by the following sequence of abstract instructions:

```
Def(counter_creator)
   NewChannel(count);                - create a new channel count
   Send(0, count);                   - send 0 to count
   NewClosure(1, closure);           - create a closure for one variable
   Store(count, closure[1]);         - put count into closure

   NewPattern(2, pattern);           - create a pattern of 2 terms
   Store(inc, pattern[1]);           - first term is the constant inc
   Store(ANY, pattern[2]);           - second term is the constant ANY
   NewRMethod(pattern, closure, _inc, inc_mth);
   Post(inc_mth, AP[1]);             - post the method to AP[1], i.e. c

   NewPattern(2, pattern);           - create a pattern of 2 terms
   Store(dec, pattern[1]);           - first term is the constant dec
   Store(ANY, pattern[2]);           - second term is the constant ANY
   NewRMethod(pattern, closure, _dec, val_mth);
   Post(dec_mth, AP[1]);             - post the method to AP[1], i.e. c

   NewPattern(2, pattern);           - create a pattern of 2 terms
   Store(value, pattern[0]);         - the first term is the constant value
   Store(ANY, pattern[1]);           - the second term is the constant ANY
   NewRMethod(pattern, closure, _val, val_mth);
   Post(val_mth, AP[1]);             - post the method to AP[1], i.e. c
   Ret();                            - terminates
End
```

The representation is derived directly from the source code. A channel `count` is created and initialized with 0. Then three methods (`_inc`, `_dec`, `_val`) are then created and are posted to the input channel `c`. For example, to create method `_inc`, a closure and a pattern are created to hold the free variable `count` and the pattern `(inc, ANY)` respectively. The method is then posted to `AP[1]` (i.e. channel `c`).

The definitions of the three methods are similar, only that of the `_inc` method is given below:

```
Def(_inc)

   NewChannel(one);
   Send(1, one);

   NewChannel(tmp);                    - create a new channel
   Add(CP[1], one, tmp);               - add count, 1 and put result in tmp

   NewClosure(2, closure)              - create a closure for 2 variables
   Store(CP[1], closure[1]);           - put count into closure
   Store(AP[2], closure[2]);           - put u into closure

   NewPattern(1, pattern);             - create a pattern of 1 term
   Store(ANY, pattern[1]);             - set the term be ANY
   NewRMethod(pattern, closure, _inc_aux, mth);
   Post(mth, tmp);                     - post the method to tmp
   Ret();                              - terminate

End

Def(_inc_aux)

   Send(AP[1], CP[1]);                 - send the message to count
   Send(AP[1], CP[2]);                 - send the message to u
   Ret();                              - terminate

End
```

In `_inc`, a new channel `tmp` is created and is used to hold the result of `Add`. A method is then created for `tmp` which takes a message from `tmp` and sends it to `count` (`CP[1]`) and `u` (`CP[2]`).

8.4 AM Interface

The interface layer is responsible for handling communications among abstract machines transparently.

To the user, communication to a remote machine is annotated by the symbol "@". For example, `foo@hal` represents a channel `foo` on machine `hal`.

However, once a connection is established between the agents on two machines (by the user or otherwise), more connections can be established as the computation evolves because channels can be passed around in channels.

For example, a machine A can export an agent `get_service`:

```
! get_service(S,R) -o
     S | append_serv -o R(append)
       & reverse_serv -o R(reverse)
       & ...
```

Then machine B can invoke `append` on machine A to concatenate two streams, `L1` and `L2`, as follows:

```
{ s,l } ^ (get_service@machine_a(append_serv,s)  *  s(A) -o A(L1,L2,L))
```

The details of the interaction between abstract machine will be discussed in section 10.4.

8.5 Interface Instructions

The following instructions are available for remote interactions:

- Open(h, cc) opens a connection to host `h`, returns `TRUE` to `cc` if successful, `FALSE` otherwise.
- Close(h) closes the connection to host `h`.

- <u>Send(m, c, h)</u> sends a message `m` to channel `c` on host `h`.
- <u>Post(m, c, h)</u> posts a method `m` to channel `c` on host `h`.

Chapter 9

Compiler

The current implementation of the compiler is primitive but it provides the infra-structure for developing the analysis for various optimizations mentioned in this chapter.

The compiler is written in C++ and is structured as follows:

1. Phase 1 (Front end): Phase one performs lexical analysis and syntax parsing to produce an abstract syntax tree representing the input program. The lexer and parser are generated from lex++ and yacc++.

2. Phase 2 (Mid section): Phase two implements transformations on the abstract syntax tree to generate another abstract syntax tree which may give better runtime characteristics.

3. Phase 3 (Back end): Phase three of the compiler takes a syntax tree, chooses a traversal order to walk the tree nodes and generates abstract machine instructions for the tree nodes.

Once the abstract instruction code is generated, another pass is used to link with the abstract instruction definitions with the runtime system to generate an executable file.

The front end and back end of the compiler are relatively straight-forward. The focus of this chapter is on the possible transformations occur during phase 2.

9.1 Optimizations

In the current implementation, each abstract machine instruction is implemented in C++, as a macro or a function. In other words, the output of the compiler is a C++ program. As a result, many of the common compiler optimizations are available when a good C++ compiler is chosen to compile the abstract machine code. However, there is another set of important (semantic/algorithmic) optimizations specific to the design of Linear Janus. Those optimizations are responsible for order of magnitude improvement in the runtime performance. Some of the more interesting ones will be discussed in this chapter[1].

In the following, we use `counter_creator` as an example to describe how transformations can apply.

9.1.1 Closures

Linear Janus is lexically-scoped. As a result, an agent may contain free names or parameters which are bound in an statically enclosing context. For example, consider the following agent, which may be defined within a larger context:

```
! fwd(R) ⊸ x(M) ⊸ R(M)
      ...
```

The name `x` is free in `fwd` and `R` is free in `x(M) ⊸ R(M)`.

The above program is compiled as follows:

[1]These optimizations have been studied by transforming the source manually or by annotating the source with compiler directives, since the mechanisms needed to infer the necessary information for our algorithm are still under development.

```
Def(fwd)

  NewPattern(1, pattern);              - create a new pattern for 1 term
  Store(ANY, pattern[1]);              - set the term to be ANY

  NewClosure(1, closure);              - create a new closure for 1 variable
  Store(AP[1], closure[1]);            - put r into the closure

  NewRMethod(pattern, closure, _fwd, mth);
  Post(mth, X);                        - post the method to X
  Ret();                                 - terminate

End

Def(_fwd)

  Send(AP[1], CP[1]);                  - send m to r
  Ret();                                 - terminate

End
```

In `fwd`, a new closure is created to hold `r`. Since there may be multiple invocations of `fwd` at the same time, a new closure is necessary for each individual invocation.

However in many situations, we can avoid creating a new closure at new closure at each invocation. Consider `counter_creator` where `count` is a free name in all three methods. Since all three methods share the same `count`, they share the same closure (in the three instances of `NewRMethod`).

```
Def(counter_creator)

   NewChannel(count);
   Send(0, count);

   NewClosure(1, closure);              - create a closure for one variable
   Store(count, closure[1]);            - put count into closure

   NewPattern(2, pattern);
   Store(inc, pattern[1]);
   Store(ANY, pattern[2]);
   NewRMethod(pattern, closure, _inc, inc_mth);     - use closure
   Post(inc_mth, AP[1]);

   NewPattern(2, pattern);
   Store(dec, pattern[1]);
   Store(ANY, pattern[2]);
   NewRMethod(pattern, closure, _dec, val_mth);     - reuse the closure
   Post(dec_mth, AP[1]);

   NewPattern(2, pattern);
   Store(value, pattern[0]);
   Store(ANY, pattern[1]);
   NewRMethod(pattern, closure, _val, val_mth);     - reuse the closure
   Post(val_mth, AP[1]);

   Ret();
End
```

However, consider the body of the three methods:

$$\mathtt{add}(\mathtt{count}, \mathtt{\hat{}1}) \mid \mathtt{N} \multimap \{\, \mathtt{count(N)} \ * \ \mathtt{R(N)} \,\}$$

$$\mathtt{sub}(\mathtt{count}, \mathtt{\hat{}1}) \mid \mathtt{N} \multimap \{\, \mathtt{count(N)} \ * \ \mathtt{R(N)} \,\}$$

$$\mathtt{count} \mid \mathtt{N} \multimap \{\, \mathtt{count(N)} \ * \ \mathtt{R(N)} \,\}$$

besides count, R is also a free variable. Hence, a new closure is created in _inc:

```
Def(_inc)

   NewChannel(one);
   Send(1, one);                        - create the channel for ^1

   NewChannel(tmp);                     - create tmp for add(count, ^1)

   Add(CP[1], one, tmp);                - add(count, ^1, tmp)

   NewClosure(2, closure)               - create a closure for 2 variable
   Store(CP[1], closure[1]);            - put count into closure
   Store(AP[2], closure[2]);            - put u into closure

   NewPattern(1, pattern);
   Store(ANY, pattern[1]);
   NewRMethod(pattern, closure, _inc_aux, mth);
   Post(mth, tmp);
   Ret();

End

Def(_inc_aux)

   Send(AP[1], CP[1]);                  - count(N)
   Send(AP[1], CP[2]);                  - R(N)
   Ret();                               - terminate

End
```

In general, a new closure is needed for each invocation because in a concurrent setting there may be multiple invocations active simultaneously and each invocation may have a different u.

However, in `counter_creator`, because of the shared access to `count`, all methods (and their multiple invocations) are serialized. Assuming that `Add` (a primitive whose behavior we can define) will block until the addition can be completed, then, a clever trick is to reuse the closure passed into `_inc` if the closure has been allocated with one more slot:

```
Def(_inc)
```

```
   NewChannel(one);
   Send(1, one);                          - create the channel for ^1

   NewChannel(tmp);                       - create tmp for add(count, ^1)

   Add(CP[1], one, tmp);                  - add(count, ^1, tmp)

   Store(AP[2], CP[2]);                   - reuse the closure

   NewPattern(ANY, pattern);
   NewRMethod(pattern, CP, _inc_aux, mth);
   Post(mth, tmp);
   Ret();

End

Def(_inc_aux)
   Send(AP[1], CP[1]);                     - count(N)
   Send(AP[1], CP[2]);                     - R(N)
   Ret();
End
```

In the optimized code, instead of creating a closure each time a method is invoked, only one closure is needed for each counter. Thus, both storage and instructions are saved.

In the above example, `counter_creator` presents a special case in which we can reuse a single closure for all invocations of all methods. In the general case, the optimization is only applicable for a method which is posted to a local and temporary channel (i.e. the channel is never used again after the method is invoked).

Instead of doing the transformation as above, an alternative solution commonly employed in lexically-scope functional language is to pass free variables as extra arguments. This latter method will work well for us, but seems to incur more overhead if the scope of a variable is deeply nested, as is common in Linear Janus.

9.1.2 Messages and Methods

Another area of inefficiency arises from the fine granularity of action performed by agents. In our previous exposition, communication happens when messages and methods combine at a channel.

This picture is conceptually simple because the relationship among a channel, a message and a method is symmetric. Moreover, the implementation is straight-forward since there is no need to have, for example, a suspension queue for methods that do not find a matching message. Unfortunately, a naive implementation of this scenario will be very inefficient.

Consider the compilation of the `_inc` method given above. A method (`_inc_aux`) is created dynamically and is posted to the newly created channel `tmp`. However, if `Add` produces the results to `tmp` before the method is posted (as is the case for us), we can avoid creating the method altogether, by simply fetching a message from `tmp` and breaking the symmetry between messages and methods as follows:

```
Def(_inc)

   NewChannel(one);
   Send(1, one);                        - create the channel for ^1

   NewChannel(tmp);                     - create tmp for add(count, ^1)

   Add(CP[1], 1, tmp);                  - add(count, ^1, tmp)

   Recv(tmp, n);                        - add(count, ^1) | N
   Send(n, CP[1]);                      - count(N)
   Send(n, AP[2]);                      - R(N)
   Ret();

End
```

Instead of posting a method to `tmp` which results in a match, a message is directly fetched from `tmp` and sent to `u` (`AP[2]`) and `count` (`CP[1]`).

The above transformation can be safely applied when it can be determined that a message resides in the channel when requested (i.e. no suspension is required). In `_inc`, this is the case, since `tmp` is local channel and `Add` is a primitive agent which we can make to block until the addition completes.

This optimization turns out to be applicable in many cases and improves the performance greatly, compared with the naive compilation. The price to pay for the improvement in performance is a more complicated implementation because special mechanisms are needed to deal with the suspension of agents when `Recv` cannot be completed (i.e. when there is no message in the channel). Currently, the runtime system is responsible for handling the failure of `Recv`. However, it would have been simpler to make the compiler responsible for creating and posting the method as in the normal case when `Recv` fails.

9.1.3 Specializing Channels

In Linear Janus channels are used for many different purposes, such as communication, synchronization and storage. While this generalization of channel provides great flexibility, it also leads to poor performance when implemented naively. To satisfy the general usage, a channel has to be implemented as a record containing multisets for messages and methods. However, many channels are used in a particular way and by analyzing the usages more carefully we can optimize the representations to fit the particular uses.

In this section, we describe three different uses of a channel and in each case we develop a better representation for it.

- **Storage Channel**: Very often a channel is used as a storage unit, since there is no other means of storage in the language. For example, `count` in `counter_creator` is used as a single element storage. A channel used as storage can be better implemented as a *storage channel*. Instead of `Send`, an object is sent to a storage channel by `ScSend`. Moreover, a storage channel is accessed only by `ScRecv` which returns the object in the channel.

  The restricted protocol allows us to implement a storage channel in a single word of memory and accesses can be done without pointer dereference (chapter 10). The small memory requirement and the faster access significantly decrease the overhead of using channels as storage elements.

  The analysis to determine if a channel is used as a singular message storage requires the following two conditions:

  1. The reachable scope of the channel is closed, i.e. all usage of the channel can be accounted for.
  2. There can be at most one message in the channel at any time. This can be determined, for example, if every consumer of the message produces a message to the same channel eventually.

  For example, `count` (in `counter_creator`), `one` and `tmp` (in `_inc`), satisfy both of the above conditions and thus can be created as a storage channel:

```
Def(counter_creator)

   NewScChannel(count);
   ScSend(0, count);

   ...
End

Def(_inc)

   NewScChannel(one);
   ScSend(1, one);                      - ^1
   NewScChannel(tmp);
   Add(CP[1], one, tmp);                - add(count, ^1, tmp)
   ScRecv(tmp, n);                      - add(count, ^1) | N
   ScSend(n, CP[1]);                    - count(N)
   Send(n, AP[2]);                      - R(N)
   Ret();

End
```

Currently, when the compiler cannot determine the full usage of a channel (e.g. when the channel is passed to another agent), it may still choose an *immediate_channel* for the channel speculatively as long as the known usage is consistent with that of a storage channel. During the runtime, a misuse will cause an exception which would convert the storage channel back to a normal channel.

Remark. Implementing a channel as a storage channel would open up further opportunities for the C++ compiler to optimize the generated abstract instructions. For example, the instructions above would generate the following C++ code:

```
Def(_inc)

   one = hp++;
   *one = 1;
   tmp = hp++;
   *tmp = *CP[1] + 1;
   n = *tmp;
```

```
    _send(n, AP[2]);
    *CP[1] = n;

End
```

which can be readily optimized by any C++ compiler into:

```
Def(_inc)

    hp++;
    hp++;
    *CP[1] = *CP[1] + 1;
    _send(CP[1], AP[2]);

End
```

- **Method Channel**: Many channels have only one method and can be determined to be closed during compile time, i.e. no new method would be added. In such case, the channel is implemented as a *method channel*. A method channel only handles incoming messages and not methods (there should be none, by definition). Instead of `Send`, `McSend` is used to send an message to a method channel.

  When a message which matches the pattern of the method arrives at a method channel, a new agent is created. However, since an unmatched message will never be handled, a method channel does not need to store them. As a result, a method channel can be implemented by a pointer pointing to the method.

  To make our optimization more applicable, methods for the same channel are coalesced into one with a case statement. For example, instead of creating three methods (`_inc`, `_dec`, `_val`) for a counter, we can instead create a single method which handles all three with the same piece of code[2].

```
Def(counter_creator)
    NewScChannel(count);
```

[2]Actually the transformation is not strictly applicable because `c` may have other methods defined elsewhere. However, we can inline the call of `counter_counter` which may close off `c`. In which case, `c` is created as a method channel and the posting of a method simply puts the pointer to the method into the channel.

```
  ScSend(0, count);

  NewClosure(1, closure);
  Set(closure[1], count);
  NewRMethod(NULL, closure, counter_method, mth);
  Post(mth, AP[1]);
  Ret();
End

Def(counter_method)
  Jne(AP[1], inc, Dec);

 Inc:
  NewScChannel(one);
  ScSend(1, one);                       - ^1
  NewScChannel(tmp);
  Add(CP[1], one, tmp);                 - add(count, ^1, tmp)
  ScRecv(tmp, n);                       - add(count, ^1) | N
  ScSend(n, CP[1]);                     - count(N)
  Send(n, AP[2]);                       - R(N)
  Ret();

 Dec:
  Jne(AP[1], dec, Value);
  NewScChannel(tmp);
  Sub(CP[1], one, tmp);                 - sub(count, ^1, tmp)
  ScRecv(tmp, n);                       - sub(count, ^1) | N
  ScSend(n, CP[1]);                     - count(N)
  Send(n, AP[2]);                       - R(N)
  Ret();


 Value:
  ScRecv(CP[1], n);                     - count | N
  ScSend(n, CP[1]);                     - count(N)
  Send(n, AP[2]);                       - R(N)

 Done:
  Ret();
End
```

All top level agents are assumed to be closed and implemented as method channels.

Besides requiring less memory, this optimization allows us to circumvent the pattern matching when the message arrives at the channel. Instead, we can immediately create a new agent within which the message is tested to find the proper handler.

- **Virtual Channel**: Many channels are created for transient purposes, such as holding a message until the message can be forwarded to another agent. Ideally, we would like to equate or unify the temporary channel with the target channel (in the same manner as unification in logic programming), thus eliminating the forwarding. However, equality or unification would create many semantic problems and implementation difficulties (especially in a concurrent setting). Instead, if the compiler determines a message is forwarded from a temporary channel to another channel, it would create the temporary channel as a *virtual channel*. The intention is that when an object is sent to the virtual channel, it will show up at the target channel, i.e. an expedited forwarding. Instead of `Send`, `VcSend` is used to send an object to a virtual channel.

  For example, consider the following example:

  $$\mathtt{x} \mathbin{\hat{}} \{\ \mathtt{div(P,Q,x)}\ *\ \mathtt{x(M)} \multimap \mathtt{Y(M,\hat{}())}$$

  Instead of creating a channel for `x`, a 2-tuple is created with a hole and `x` as a virtual channel points to the hole in the 2-tuple. When the result is produced on `x`, it fills the hole in the 2-tuple.

In order to implement the various kinds of channels mentioned above, new abstract instructions are introduced. We retain `Send` and `Post`, which serve as the catch-all instructions, i.e. they handle all kinds of channels, at a slower rate however. If the type of a channel is known, one of the specialized instructions can be used to expedite the operation.

Besides the special `Send`'s and `Recv`'s, new instructions for various kinds of channels are also introduced and are included in the tables in appendix C.

9.1.4 Remark

As mentioned before, the abstract machine instructions are implemented as C++ macros and inlined functions. Given this, we can implement agents in two different approaches. In the first one, all agent definitions (e.g. `append` and `counter_creator`) are compiled into a single C++ program with GOTO labels for different agents. In the second approach, which we have adopted, each agent is compiled into an individual C++ function. The former scheme may potentially be more efficient because a GOTO is usually cheaper than a procedure call. However, the latter scheme seems more flexible, since it allows us to compile each agent independently and to dynamically load or unload them individually. To reduce the overhead of procedure calls, functions which represent agents are declared as inlineable to allow the compiler to inline the agent definitions when appropriate.

Chapter 10

Runtime System

The runtime system implements the mechanisms and instructions of the abstract machine. As discussed in previous chapters, the abstract machine is implemented in C++ and each abstract machine instruction is either a macro or an inline function, thus allowing us to "compile" an agent, instead of interpreting an agent during runtime. This approach not only allows us to prototype the abstract machine and the compiler quickly, but also produces better runtime performance since we can leverage optimizations of the C++/C compiler.

10.1 Runtime Model

There is plenty of flexibility in the runtime implementation of the abstract machine, since the definitions given in chapter 8 make few assumptions about the implementation.

The overall operation of the runtime system is as follows. The runtime system maintains a queue and a heap. The queue contains agents which are ready for execution[1] and the heap holds data structures which are allocated dynamically. Upon startup, the queue, the heap and other bookkeeping information (see below) are initialized. After the initialization, the *scheduler* is invoked. The scheduler dequeues an agent from the queue and executes it. During the execution of the agent, new agents may be created and enqueued as a result of sending messages and posting methods. Eventually, when the agent terminates, control returns to the scheduler which then tries to execute another agent from the active queue.

The full detail of the runtime system is described in a document in preparation. In the following, we will concentrate on some of the important aspects of the runtime implementation.

[1] There is no suspension queue, since all activities are suspended at the corresponding channels.

10.2 Data Format in the runtime system

Each object in the runtime system is either a word (4 bytes) or a record (of arbitrary size). Each word has a 2-bit (LSB) tag indicating the type of the word[2]. The four different tags are:

- Pointer Tag: The word represents a normal pointer.
- Forward Pointer Tag: The word represents a forward pointer. A forward pointer is used during garbage collection to point to the new location of the moved word (section 10.3.2).
- Descriptor Tag: The word represents an object descriptor.
- Immediate Tag: The word represents an immediate object, e.g. an integer.

In our tag scheme, pointer has tag b'00 and immediate object has tag b'01. This scheme makes pointer operations efficient (because C pointers can be used directly) while some overhead in integer operations is incurred (because the tags need to be dealt with). However, given that pointer operations are more common than integer operations, making the former more efficient gives a better result in general.

An object that can be represented by 30 or less bits (such as an integer) occupies one word, other objects are allocated records of the right size. For a record, the first word is always the descriptor. The descriptor consists of the type of record (3 bits for the type, and 1 bit for remote/local) as well as the length of the record (26 bits):

```
31---------------------6--------------------2-----0
| length of the record | type of the record | TAG |
---------------------------------------------------
```

The types of records are:

- LChannel: a local channel
- RChannel: a remote channel

[2]The lower 2 bits are used because on SPARCStations, a word is aligned at word boundary, i.e. the lower two bits of a pointer are always 0.

- RMethod: a reusable method
- NMethod: a nonce method
- Message: a reusable message

10.2.1 Messages, Methods, and Channels

Multisets of messages and methods in a channel are implemented as doubly-linked lists.

A message is a record with the following slots:

1. Next: pointer to the next message in the linked list.
2. Prev: pointer to the previous message in the linked list.
3. Message: content of the message.

Similarly, a method is defined with the following slots:

1. Next: pointer to the next message in the linked list.
2. Prev: pointer to the previous message in the linked list.
3. Code: pointer to the body of the method (a C++ function).
4. Pattern: pattern of the method.

Finally, a channel is record with two slots:

1. Methods: pointer to a linked list of methods.
2. Messages: pointer to a linked list of messages.

10.2.2 Agent Representation

Each top level agent (and each direct channel) is compiled into a C++ function. The `Code` pointer of a method object points to the code which represents the body of the method (i.e. an agent).

When an agent is created (i.e. when a message matches a method), a process record is allocated with the message and the method. When the process record is executed, the message is used as the argument to call the C++ function pointed by `Code` in the method.

10.2.3 Internal Data Structures

Besides the above data structures, the runtime system also keeps the following internal data structures:

- Process Record: Each runable process (agent) is represented by a process record. A process record consists of a message and a method. When a process record is executed, the message is used as the argument to invoke the function pointed by `Code` in the method, i.e. `((void *) method.Code)(message)`.
- Process Queue: Process queue is implemented by a C++ class, which exports the following interface:

  ```
  int init(): initialize a queue
  int empty(): return True if the queue is empty
  Pr *dequeue(): return a record from the queue
  int enqueue(Pr *): insert a record into the queue
  int execute(Pr *): execute a process
  ```

  Given a process queue `pq`, the scheduler is defined by the following loop:

  ```
  while(!pq.empty())
    pq.execute(pq.dequeue());
  ```

- Hash Table: For bookkeeping purposes, the runtime maintains a number of tables, each as a hash table. The hash table class exports the following interface:

  ```
  init():  initiation the hash table
  insert(Key, item): insert item with key Key
  lookup(Key): lookup the entry which matches Key
  delete(Key): delete the entry which matches Key
  ```

10.3 Memory Management

Objects created during runtime is allocated on a heap. In the implementation, the heap is simply a region of the data segment of the runtime system (which is a normal Unix process). Two pointers, `heapLo` and `heapHi`, are used to keep track of the limits of the current space. A third pointer, `hp`, points to the first unused location in the current space. When *n* words are requested from the heap, the contents of `hp` is returned and then incremented by *n*.

For example:

`NewRMethod(pattern, closure, code, result)`

can be implemented by the following instructions:

```
result = hp;
hp += sizeof(Method);
if (hp > heapHi) gc(result);
((Method) result).desc    = RMethod;
((Method) result).pattern = pat;
((Method) result).closure = closure;
((Method) result).code    = code;
```

10.3.1 Fast Allocation

Since memory allocations happen very often, it is important that the allocation is as fast as possible. The above instructions sequence for `NewRMethod` seems to be optimal. However, a trick is to avoid checking the heap limits before/after each allocation. Instead, when the heap limit is exceeded, an exception is generated. The exception handler then invokes the garbage collector. The rationale is that the heap limit is exceeded only when memory is exhausted which does not happen very often, and when it does we have to reclaim the memory, a lengthy process that would make the extra overhead of exception handling insignificant.

This scheme is implemented by fooling the protection for data segment in SunOS. Using *sbrk()*, the limit of the data segment is set to the limit of current heap space. Allocations return without limit checking (e.g. without the `if` statement in the instructions for `NewRMethod`). However, when a word is written to a memory location outside the heap space (i.e. outside the data segment), a segmentation fault is generated. Our segmentation fault handler then invokes the garbage collector before resuming the execution of the program.

By eliminating the heap limit check, this optimization speeds up memory allocations by 5 to 20 percent, depending on the length of the allocated objects.

10.3.2 Local Garbage Collection

Objects allocated in the heap are not destroyed explicitly. To reclaim memory occupied by obsolete objects, garbage collection is performed when the heap space is exhausted. Our garbage collection is implemented by a stop-and-copy algorithm.

The stop-and-copy algorithm requires the heap to be partitioned into two semi-spaces, commonly referred to as *from_space* and *to_space*. Objects are allocated from the *from_space* until it is exhausted. Then objects that are reachable from a *root* set are copied to the *to_space*. Finally, the roles of *from_space* and *to_space* are switched.

The garbage collector is very simple because it only needs to understand the four types of tags. The root set in our garbage collection is the process queue. When a word or record is copied from *from_space* to *to_space*, a forward pointer pointing to the new location in *to_space* is written to the original location in *from_space*, thus the garbage collector can re-direct later references to the new location.

10.3.3 Efficiency of Garbage Collection

The efficiency of a stop-and-copy garbage collector is determined by the amount of live data at the time of the collection. For many applications, such as `append` and `naive-reverse`, most of memory would become non-reachable at the point of garbage collection, so the overhead of garbage collection is small. However, for applications in which objects persist for a long time, a generational garbage collector can reduce the overhead of copying the persistent objects. Alternatively, the compiler could use information on the persistence of an object and allocate objects from one of the many spaces. However, both alternatives require significant effort in the implementation and have not yet been pursued.

10.3.4 Global Garbage Collection

Since abstract machines can communicate with each other, some references may be from other abstract machines. It is inefficient and infeasible if all abstract machines have to garbage collect at the same time. Instead, we use a counting scheme to track references to and from other abstract machines and each abstract machine can perform its own garbage collection independent of others. To implement the scheme, each abstract machine keeps two tables:

- Outgoing Reference Table (ORT). The ORT keeps track of references to objects on other abstract machine. When a remote object is received, an entry is created for the object and is inserted into the ORT. An entry in the ORT consists of:
  - host_id: the identifier for the home node of the object.
  - object_id: a unique identifier to locate the object at its home node.
  - reference_count: the number of references to the object.
- Incoming Reference Table (IRT). The IRT keeps track of references from other machines to local objects. During garbage collection, the IRT is used together with the process queue as the root set. Each entry in the IRT consists of:
  - object_id: an unique identifier to locate the object.
  - reference_count: the number of references to the object.
  - object_pointer: the local pointer to the object.

During local garbage collection, the IRT is used to complement the process queue as the root set, i.e. an object is not garbage when it is reachable from the process queue or has a remote reference. When an outgoing reference is canceled during garbage collection, a `reference_cancel` message is sent to the home of the object which decrements the reference count of the object in its IRT and if the reference count reaches 0, the entry is deleted.

If local garbage collection repeatedly fails due to remote references, a *gc_request* is broadcasted to machines that hold those references. If garbage collection still fails after a *gc_request*, the heap space is expanded by allocating a larger data segment or the runtime system exits reporting memory exhaustion.

In the current implementation, the actual number of reference is recorded in the reference tables. So each time a reference to an object is canceled, a `cancel_reference` message is sent to the home of the object. Moreover, if an object from machine H is forwarded by machine A to machine B, A would retain references to the object in both of its IRT and ORT, even though it may never need the object itself. When machine B cancels the reference to the object, it sends a `cancel_reference` message to A. Eventually, machine A sends a `cancel_reference` message to H to cancel the reference.

There exists more efficient methods to implement the reference counting across multicomputers using *weights* [16], which require substantially less communications (e.g. machine B would send the cancel request to H directly in the above example). However, we have used a simple-minded implementation in which the actual number of references is recorded. As a result, the overhead of cross machine communication is rather high in our implementation.

10.4 Remote Interface Implementation

Another layer in the implementation of the abstract machine deals with issues of interfacing with agents on other abstract machines. Communications between abstract machines (which can reside on the same machine or on different machines) are implemented using Unix sockets. Each abstract machine maintains a standard input port which listens for communication requests from other machines, as well as a standard output port which sends communication requests to other machines.

For example, if an agent on machine A sends a message to an agent on machine B, machine A sends a connection request to the standard input port of machine B which acknowledges the request by opening a new socket connection for the communication. Then the agent on machine A will use the new connection for communication with the agent on machine B.

Since the handshake required to open a new socket connection is significant when there is a lot of communicating agents on different machines, the runtime system maintains a host table (HT) listing all connections that have been established. When a connection to a machine is requested, HT is consulted. If the connection already exists, it is reused. Otherwise a new one is established and HT is updated. Remote machines to which a machine has a connection are called its *acquaintances*.

In addition, agents can enter and exit a machine dynamically and system configuration can change over time, as in an open system. Hence a service table (ST) is kept by every machine listing the exported services that other machines can request. When a certain service is needed, a query is sent to one or more *acquaintances* of the machine. A positive acknowledgment will be returned by the machine which provides the service.

10.4.1 Remote and Local References

When an object is sent to a remote machine, it is scanned and pointers to local objects are changed to be *remote pointers*. A remote pointer consists of a *host_id*, which uniquely identifies the host machine, and an *object_id* which uniquely identifies an object on a machine. The IRT is updated accordingly. On the remote end, when an object is received, it is scanned for any remote pointers and the ORT is updated to reflect any reference to objects on other machines.

For example, if `Send` is executed with a remote channel (i.e. a channel that resides remotely), the message is marshaled as mentioned above and is sent to the remote machine. The process is transparent to the programmer.

As an example, consider `get_service` below:

```
! get_service(S,R) —o
     S | append_service —o R(append)
       & reverse_service —o R(reverse)
       & ...
```

If machine B wants to get `append` service from machine A to append two streams `L1` and `L2`, it would create an agent:

```
{ s,l } ^ { get_service@machine_a(append_service,s) * s(A) —o A(L1,L2,l) }
```

which creates two channels `s` and `l` and send `append_service` and `s` to machine A. On machine A, `get_service` receives the message containing `append_service` and a channel bound to `R`. When machine A sends `append` to `R`, the runtime system will find out `R` is a remote channel and will send `append` to `s` on machine B. When machine B receives the message from machine A, it searches its IRT for the current location of `s` (which may have been relocated during garbage collection) and sends the message to `s`. Similarly, the two lists would then be sent to machine A and etc.

In our example, many data are moved between machine A and B in doing the remote append. Compounded with the long latency and high overhead of Unix sockets, the remote append would perform miserably. This can be improved by migrating, for example, the lists to where `append` is or vice versa. This work surely is an area of research that can be exploited.

Chapter 11

Remarks on the Implementation

The goal of developing the current prototype system is to understand issues in implementing a language like Linear Janus. The current implementation is still at an early stage. While the runtime is quite efficient, the compiler is currently incapable of performing the optimizations mentioned in chapter 9 automatically.

On the other hand, the prototype system has provided us a platform to investigate and understand many issues. Our preliminary experiments show that unifying the communication and synchronization using channels are not prohibitively expensive. In fact, compared with Linda, communication using channels is both more efficient and easier to implement (especially in a distributed environment) because unlike a tuple space, each channel has a selected set of readers and writers.

In addition, by optimizing representations for channels (as described in chapter 9), our runtime performance compares favorably with the same programs compiled and run on language systems that are close in nature to Linear Janus, e.g. Prolog and Lisp. For example, to compare the runtime performance of `append` and `nrev` with the same programs written in Prolog and Lisp, we manually optimized the code from our compiler using the optimizations discussed in chapter 9. The table below presents the timing statistics (in msec) of `append` and `nrev` in Prolog, Lisp and Linear Janus[1] (the source codes are presented in appendix D).

Moreover, by connecting multiple abstract machines to form a network, we have a mechanism to combine multiple workstations to form a powerful computing engine, with few extra syntax in the

[1] All programs are compiled and run on a SPARCStation 2 with 32 megabytes of memory. All timings exclude the system startup time and each system allocates five megabyte of heap storage. The numbers are the averages of 10 consecutive runs.

	Sictus Prolog (v.1.8)	Lucid Common Lisp (v.4.1)	Linear Janus
append 10000	49	210	55
append 100000	509	stack overflow	556
nrev 1000	2579	6830	2723

Table 11.1: Performance data for Prolog, Lisp and Linear Janus.

source language. Unfortunately, we have yet to develop any extensive application that will exploit the available networking capability.

Needless to say, there is still plenty left to be done. We are still finding and trying new optimizations. For example, we have tried to use a compiler managed stack to execute asynchronous agents as procedure call and recursive methods (such as `append`) as recursive procedure calls. Initial data indicates that this optimization can reduce the running time of `nrev 1000` by nearly 50 percent (to 1399 msec). In the runtime system, we can also fine-tuning the performance by allocating important abstract machine registers in hardware registers and by optimizing codes that are commonly used.

While the implementation of the compiler and runtime system will improve over time, the big jump in performance is likely to be brought forth by something more fundamental. For example, we need a linear type system that will provide the information that is needed for the optimizations discussed in chapter 9. There have been a number of studies using linear types ([2], [23]) to determine the storage requirement of an object, thus allowing a more efficient memory management. Another possible analysis that can improve the runtime performance significantly is to determine the dependence among agents, so that we can schedule the producer of a message before the consumer to avoid undue suspension.

Bibliography and references

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs* The MIT Press, 1985.

[2] S. Abramsky. Computational Interpretation of Linear Logic. Technical report, DOC, Imperial College, London, U.K., 1990.

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *Computer*, Vol. 19, No. 8, Aug. 1986, pages 26–34.

[4] G. Agha. *Actors: a model of concurrent computation in distributed systems*. PhD thesis, University of Michigan, 1985.

[5] Gil Agha and Carl Hewitt. Concurrent Programming Using Actor. *Object-Oriented Concurrent Programming*, ed. Akinori Yonezawa and Mario Tokoro, MIT Press, 1987.

[6] J.W. de Bakker and J.N. Kok. Uniform abstraction, atomicity and contractions in the comparative semantics of Concurrent Prolog. In *Proceedings of the Fifth Generation Computer Systems Conference*, December 1988.

[7] Paul S. Barth, Rishiyur S. Nikhil and Arvind. M-structures: Extending a Parallel, Non-strict, Functional Language with State. MIT LCS Computation Structures Group Memo 327, March 1991.

[8] J.A. Bergstra and J.W. Klop. *Algebra of Communicating Processes with Abstraction* Journal of Theoretical Computer Science, Vol 33,, pp 77–121, 1985.

[9] G. Berry and G. Boudol. The Chemical Abstract Machine. In *Proc.* 17^{th} *ACM Conf. on Principles of Programming Languages*, pages 81–94, 1990.

[10] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, July 1984.

[11] K. L. Clark and S. Gregory. PARLOG: parallel programming in logic. *TOPLAS*, 8(1):1–49, January 1986.

[12] W.L. Clinger. *Foundations of Actor semantics*. PhD thesis, MIT, May. 1981.

[13] Williams J. Dally *et al*. The J-Machine: A Fine Grain Concurrent Computer. In *Proceedings of the IFIPS 1989 Conference.*

[14] I. Foster and S. Taylor. Strand: A Practical Parallel Programming Language. In *North American Conference on Logic Programming*, pages 497–512. MIT Press, October 1989.

[15] Ian Foster and Steven Tuecke. Parallel Programming with PCN. Technical Report ANL-91/32 Version 1.2, December 11, 1991.

[16] Ian Foster, Steven Tuecke and Stephen Taylor. A Portable Run-Time System for PCN. Technical Report ANL/MCS-TM-137, July 1991.

[17] J.Y. Girard. Linear Logic. Theoretical Computer Science 50 (1987), page 1–102. Elsevier Science Publishers B. V. (North-Holland).

[18] David Gelernter. Multiple Tuple Spaces in Linda. In *Proceedings of PARLE '89*, vol.2, pp. 20–27, 1989

[19] G.A. Gunter and V. Gehlot. Net as Tensor Theory. In G. De Michelis, editor, *Proc. 10-th International Conference on Application and Theory of Petri Nets, Bonn*, pp. 174–191, 1989.

[20] Sussanne Hupfer. Melinda: Linda with Multiple Tuple Spaces. Technical Report YALEU/DCS/RR-766, Department of Computer Science, Yale University, 1990.

[21] S. Jagannathan. Optimizing Analysis for First-Class Tuple-Spaces In *Advances in Language and Compiler for Parallel Processing*, Alexandru Nicolau, David Gelernter, Thomas Gross and David Padua (Editors), MIT Press, 1991.

[22] N. Kobayashi and A. Yonezawa. ACL - A Concurrent Linear Logic Programming Paradigm. In *Proceedings of the International Logic Programming Symposium*, ed. D. Miller, pp. 279 – 294, MIT Press, 1993.

[23] Y. Lafont. The Linear Abstract Machine. *Theoretical Computer Science*, 59(1,2), pp.157–180, 1988.

[24] C.E. Hewitt. Viewing Control Structures as Patterns of Passing Messages. In *Journal of Artificial Intelligence*, Vol. 8 No. 3 (June 1972) pp. 323–364.

[25] C.E. Hewitt. Concurrency in Intelligent Systems *AI Expert*, 1986

[26] K. Honda and M. Tokoro. *An Object Calculus for Asynchronous Communication Semantics*, M. Tokoro, editor, Object-based Concurrent Computing, LNCS, Springer-Verlag, 1992.

[27] Kenneth M. Kahn, and William A. Kornfeld. Money as a Concurrent Logic Program Xerox Palo Alot Research Center, Technical Report Series, SSL-89-22, 1989.

[28] Joxan Jaffar and Jean-Louis Lassez. *Constraint Logic Programming*. Technical Report, Monash University, 1986.

[29] Sverker Janson and Johan Montelius. Design of the AKL/PS 0.0 sequential prototype implementation of the Andorra Kernel Language. Internal PEPMA report, Swedish Institute of Computer Science, September 1991.

[30] R. A. Kowalski. Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pp. 569–574, Stockholm, 1974

[31] Ken M. Kahn and Vijay A. Saraswat. Actors as a Special Case of Concurrent Constraint Programming. *OOPSLA*, October 1990.

[32] John W. Lloyd. *Foundations of Logic Programming*. *Symbolic Computation series*, Springer Verlag, 1984.

[33] Remco Moolenaar and Henk Van Acker. A parallel implementation of AKL. Technical report, Computer Science Department, Katholieke Universiteit Leuven, 1991.

[34] Carl R. Manning. *The Design of a Core Actor Language and its Compiler*. Master's thesis, MIT, August 1987.

[35] Robin Milner. A Theory of Type Polymorphism in Programming. *JCSS 17*, 3(1978), pp. 348–375.

[36] Robin Milner. Lectures on a calculus for communicating systems. 1984. Unpublished lecture notes.

[37] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Process, Part I LFCS Report Series, 1989, LFCS.

[38] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Process, Part II LFCS Report Series, 1989, LFCS.

[39] Robin Milner Functions as Processes In M. S. Paterson, editor, *The Seventeeth International Colloquium On Automata Language And Programming*, pp. 167–180, Springer-Verlag, 1990, Lecture Notes In Computer Sceince 443.

[40] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML* MIT Press, Cambridge, Mass., 1989

[41] Rishiyur Nikhil. *ID Reference Manual (Version 88.0).* Technical Report, MIT, 1988, Computation Structures Group Technical Report.

[42] Rocco De Nicola and Matthew Hennessy. CCS without τ's. In *TAPSOFT 87, Lecture Notes In Computer Science 249*, pp. 138–152, Springer-Verlag, 1987.

[43] John F. Palmer. The NCUBE Family of Parallel Supercomputers. *Proceedings IEEE International Conference on Computer Design, ICCD-86*.

[44] S. L. Peyton Jones *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science, 1987.

[45] G. D. Plotkin *A structural approach to operational semantics.* Technical Report DAIMI FN-19, CS Department, University of Aarhus, 1981.

[46] W. Reisig. Petri Nets. EATCS Monographs on Theoretical Computer Science, eds W. Brauer, G. Rozenberg, A. Salomaa, Springer Verlag, 1983.

[47] John H. Reppy *High-Order Concurrency.* Ph.D Thesis, Department of Computer Science, Cornell University, June 1992,

[48] P. Roussel Prolog: Manuel de Reference et d'Utilisation. Technical Report, University of d'Aix-Marseille, Groupe de IA, 1975.

[49] Vijay A. Saraswat. Atomic Herbrand as a logic programming language. 1989. Forthcoming. Treats the downward completeness results for the cc-id language instantiated over a slightly restricted version of the Herbrand constraint system.

[50] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. To appear, Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1990.

[51] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. *Janus*: A step towards distributed constraint programming. In *Proceedings of the North American Conference on Logic Programming*, October 1990.

[52] Vijay A. Saraswat and Partick Lincoln Higher-order, linear, concurrent constraint programing. Unpublished paper, July 1992.

[53] Vijay A. Saraswat and Clifford S. C. Tse Linear distributed constraint programming. Forthcoming, 1992.

[54] Ehud Shapiro Concurrent Prolog: A progress report. *IEEE Computer*, **19**(8), pp. 44–58, August 1986.

[55] Ehud Shapiro A Subset of Concurrent Prolog and Its Interpreter *Concurrent Prolog, Collected Papers*, Eduh Shapiro editor, pp. 35, 1987

[56] Guy L. Steele Jr. Lambda: The Ultimate Declarative. Technical Report AI-TM 739, MIT AI Laboratory, June 1976.

[57] Guy L. Steele Jr. *Rabbit: A Compiler for Scheme*. Master's Thesis, MIT, 1978.

[58] David A. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In *1985 Proceedings on Functional Programming Languages and Computer Architectures*, pp. 1–16 Springer-Verlag, September 1985. Lecture Notes on Computer Science, Number 201.

[59] David A. Turner. A new implementation technique for applicative languages. In *Software – Practice and Experience*, Vol. 9, pp. 31–49, 1979.

[60] K. Ueda. Guarded Horn Clauses. Technical Report TR103, ICOT Technical Report, June 1985.

Appendix A

Intuitionistic Linear Logic

Identity Group:

$$(Identity) \qquad A \vdash A$$

$$(Cut) \qquad \frac{\Gamma \vdash A \qquad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}$$

Structural Group:

$$(Exchange) \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}$$

Logical Group:

$$(\multimap_L) \qquad \frac{\Gamma \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C} \qquad\qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad (\multimap_R)$$

$$(\otimes_L) \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} \qquad\qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad (\otimes_R)$$

$$(\&_L) \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \& B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \& B \vdash C} \qquad\qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B} \qquad (\&_R)$$

$$(!\, D) \qquad \frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B} \qquad\qquad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B} \qquad (!\, W)$$

$$(!\, C) \qquad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \qquad\qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A} \qquad (!\, S)$$

$$(\mathbf{1}_L) \qquad \frac{\Gamma \vdash B}{\Gamma, \mathbf{1} \vdash B} \qquad\qquad \vdash \mathbf{1} \qquad (\mathbf{1}_R)$$

Quantifiers:

$$(\exists_L \dagger) \quad \frac{\Gamma, A \vdash B}{\Gamma, \exists x.A \vdash B} \qquad\qquad \frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x.A} \quad (\exists_R)$$

$$(\forall_L) \quad \frac{\Gamma, A[t/x] \vdash B}{\Gamma, \forall x.A \vdash B} \qquad\qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A} \quad (\forall_R \dagger)$$

$\dagger x\ is\ not\ free\ in\ the\ lower\ sequent.$

Appendix B

λ-Compiler and λ-Evaluator

In this chapter, we discuss an encoding of λ-calculus in Linear Janus and an implementation of λ-calculus in our system. In section B.1, we first discuss an encoding of lazy λ-calculus in Linear Janus and in section B.2, we show a scheme for translating λ-expressions into combinatory terms and in section B.3, we sketch a parallel evaluator for λ-calculus using techniques of graph reductions.

B.1 Lazy λ-calculus

λ-calculus is a very simple calculus for manipulating functions. Despite its simplicity, it provides a remarkably rich model for sequential computations. In this section, we show an accurate encoding of λ-calculus in Linear Janus.

λ-calculus is formally defined as follows. We let u range over an infinite set $\mathcal{V}$ of variables and let M and N range over the set Λ of λ-terms defined as:

$$\begin{array}{lclll} M & ::= & & u & -atom \\ & & \mid & \lambda u.M & -abstraction \\ & & \mid & MN & -application \end{array}$$

The lazy reduction $\xrightarrow{\lambda}$ is defined as the smallest relation closed under the following two rules:

$$(\lambda u.M)N \xrightarrow{\lambda} M[N/u]$$

$$M \xrightarrow{\lambda} M' \qquad \Rightarrow \qquad MN \xrightarrow{\lambda} M'N$$

Encoding of λ-calculus

The encoding of λ-calculus in Linear Janus is quite simple once we realize that function applications have to map to communications. Each λ-term is translated to an agent that has a communication channel open to others. From this channel, the agent either receives inputs from other agents or outputs result to other agents. In particular, an abstraction is translated to an agent that receives inputs when it is applied, whereas an application is translated to an agent that sends to the operator a channel pointing to the operand. To define the encoding, we write $[\![\theta]\!]x$ as a map from λ-term to agents. Informally $[\![\theta]\!]x$ may be regarded as the agent that produces on x a Linear Janus *representation* of θ. A different way to interpret the notation is to say that an agent encoding M receives/sends its input/output at channel x. For convenience, we assume $\mathcal{V}$ is a subset of $\mathcal{N}$ (the set of channel names) and let r, x, y and z range over names in $\mathcal{N} - \mathcal{V}$ (to avoid variable capturing). The full encoding is defined as follows:

$$\begin{aligned}
[\![u]\!]r &= u(r) \\
[\![\lambda u.M]\!]r &= r(u, X) \multimap [\![M]\!]X \\
[\![MN]\!]r &= \{y, z\}\hat{}\{[\![M]\!]y \ * \ y(z, r) * !\ z(X) \multimap [\![N]\!]X\}
\end{aligned}$$

For instance, consider the encoding of $(\lambda u.u)N$:

$$[\![(\lambda u.u)]\!]r = r(X, Y) \multimap Y(X)$$

Assuming u is not free in N, we can obtain the following encoding for $(\lambda u.u)N$:

$$\begin{aligned}
[\![(\lambda u.u)N]\!]r &= \{x, y\}\hat{}\{[\![\lambda u.u]\!]x \ * \ x(y, r) * !\ y(Z) \multimap [\![N]\!]Z\} \\
&= \{x, y\}\hat{}\{x(X, Y) \multimap Y(X) \ * \ x(y, r) * !\ y(Z) \multimap [\![N]\!]Z\} \\
&\rightarrow \{x, y\}\hat{}\{y(r) * !\ y(Z) \multimap [\![N]\!]Z\} \\
&\rightarrow \{x, y\}\hat{}\{[\![N]\!]r * !\ y(Z) \multimap [\![N]\!]Z\} \\
&\leadsto [\![N]\!]r
\end{aligned}$$

The final step in the above is legitimate because $!\ y(Z) \multimap [\![N]\!]Z$ is inactive, since no agent has access to channel b.

The translation may look a bit peculiar. Intuitively, one may argue that the agent for λ-abstraction should be reusable since an abstraction can be applied repeatedly. In the encoding given above, however, a term is made reusable only when it actually is used (when it is in an application). Another point of interest is that laziness is preserved in the translation. The last agent in the encoding of application (! $y(Z) \multimap [\![N]\!]Z$) delays the activation of N until another agent sends N a message, which happens only when the N is applied.

The ability to encode λ-calculus within a subset of Linear Janus may seem surprising. Linear Janus is a first-order system where only accesses to channels are communicated, whereas functions are the canonical forms in λ-calculus which can be freely manipulated as any value. However, as stated by Milner[39], functional languages can be successfully implemented on machines by manipulating pointers to functions instead of functions themselves.

Finally, note that the encoding is faithful and preserves the Church-Rosser property.

B.2 λ-Compiler

In this section, we develop a compilation scheme for translating λ-terms into combinatory terms. Such a scheme allows us to exploit the variety of optimization developed for combinators and also allows us to develop an evaluator for λ-terms quite easily.

The compilation scheme is very simple and is frequently used in the compilation of functional languages (such as SASL and Miranda). The idea is to translate all λ-terms to combinatory terms, which allows many optimizations[44, 59] to be carried out.

With slight abuse of the notation, we let M, N and P range over combinatory terms, which are built from combinators by a binary (left associative) operation called applications written as MN. For our purpose, we choose the three basic combinators S, K, I and define the syntax of our combinatory logic as below:

$$M \quad ::= S \mid K \mid I \mid MN$$

Our compilation is defined by the function $\lambda_c[\![.]\!]$ (lambda compilation) which takes a λ-term and returns a combinatory term. For λ-abstractions, $\lambda_c[\![.]\!]$ uses the auxiliary function $\lambda_a\ u\ [\![.]\!]$ (λ-abstraction of u) which takes a λ-term or a combinatory term as input and returns a combinator term. The definitions of $\lambda_c[\![.]\!]$ and $\lambda_a\ u\ [\![.]\!]$ are as follows, where u, v range over $lambda$-variables in $\lambda_c[\![.]\!]$ and over $lambda$-variables or the combinatory terms S, K, I in $\lambda_a\ u\ [\![.]\!]$:

$$
\begin{aligned}
\lambda_c[\![MN]\!] &= (\lambda_c[\![M]\!]\lambda_c[\![N]\!]) \\
\lambda_c[\![\lambda u.M]\!] &= \lambda_a\ u\ [\![\lambda_c[\![M]\!]]\!] \\
\lambda_c[\![u]\!] &= u
\end{aligned}
$$

$$
\begin{aligned}
\lambda_a\ u\ [\![MN]\!] &= S(\lambda_a\ u\ [\![M]\!])(\lambda_a\ u\ [\![N]\!]) \\
\lambda_a\ u\ [\![u]\!] &= I \\
\lambda_a\ u\ [\![v]\!] &= Kv, u \neq\ v
\end{aligned}
$$

For example, consider the following compilation of $\lambda x \lambda y.xy$:

$$
\begin{aligned}
\lambda_c[\![\lambda x \lambda y.xy]\!] &= \lambda_a\ x\ [\![\lambda_c[\![\lambda y.xy]\!]]\!] \\
&= \lambda_a\ x\ [\![\lambda_a\ y\ \lambda_c[\![xy]\!]]\!] \\
&= \lambda_a\ x\ [\![\lambda_a\ y\ [\![xy]\!]]\!] \\
&= \lambda_a\ x\ [\![S(\lambda_a\ y\ [\![x]\!])(\lambda_a\ y\ [\![y]\!])]\!] \\
&= \lambda_a\ x\ [\![S(Kx)I]\!] \\
&= S(\ \lambda_a\ x\ [\![S(Kx)]\!])(\lambda_a\ x\ [\![I]\!]) \\
&= S(S(\lambda_a\ x\ [\![S]\!])(\lambda_a\ x\ [\![Kx]\!]))(KI) \\
&= S(S(KS)(S(\lambda_a\ x\ [\![K]\!])(\lambda_a\ x\ [\![x]\!])))(KI) \\
&= S(S(KS)(S(KK)I))(KI)
\end{aligned}
$$

Combinatory reductions are governed by the following three rules:

$$
\begin{array}{lrcl}
(S - reduction) & SMNP & \rightarrow & MP(NP) \\
(K - reduction) & KMN & \rightarrow & M \\
(I - reduction) & IM & \rightarrow & M
\end{array}
$$

With the above reduction rules, we can apply the combinatory term to two terms P and Q as follow:

$$
\begin{aligned}
\lambda_c[\![(\lambda x \lambda y.xy)]\!]PQ &= S(S(KS)(S(KK)I))(KI)PQ \\
&= (S(KS)(S(KK)I))P(KIP)Q \\
&= S(KS)(S(KK)I)PIQ \\
&= (KS)P((S(KK)I)P)IQ \\
&= S((KKP)(IP))IQ \\
&= S(KP)IQ \\
&= (KP)Q(IQ) \\
&= PQ
\end{aligned}
$$

To build the compiler, all we need to do is to implement $\lambda_c[\![.]\!]$ and $\lambda_a\ u\ [\![.]\!]$ to produce a combinatory representation of a λ-term. Assuming that an input λ-term is represented as a tuple, we define λ_c and λ_a as follows:

```
! λc(E,R) ⊸ E | U ⊸ R(U)
               & app,M,N ⊸ R(λc(M),λc(N))
               & abs,U,M ⊸ λa(U,λc(M),R)

! λa(U,E,R) ⊸ E | V ⊸ { eq(U,V) | true ⊸ R(i)
                                  & false ⊸ R(k,V) }
                 & M,N ⊸ R(s,λa(U,M),λa(U,N))
```

For example, $(\lambda x. + xx)$ can be represented as $\hat{}(abs, \hat{}x, \hat{}(app, \hat{}(app, \hat{}+, \hat{}x), \hat{}x))$ and compiled to $\hat{}(s, \hat{}(s, \hat{}(k, \hat{} +), \hat{}i), \hat{}i)$.

Once the a λ-term is compiled into combinatory terms, it can be subjected to a whole variety of optimizations[59, 44]. For example, consider the following reduction:

$$S(KM)(KM)N \xrightarrow{S} KMN(KMN) \xrightarrow{K} M(KMN) \xrightarrow{K} MM$$

It can be shown that $S(KM)(KN)$ can always be optimized to MN. We will not get any further into the details of those optimizations.

B.3 λ-Evaluator

An important reason for the wide use of combinators to encode λ-calculus is that combinatory reductions are extremely simple and can be carried out very efficiently. In this section, we show how to implement a parallel λ-evaluator using combinatory reductions.

The evaluator translates a λ-term into a combinatory term as described in the last section. The combinatory term is then converted into a combinator graph which can be reduced using graph reduction rules to be defined below. A combinator graph is a graph in which combinators are represented by leave nodes and applications MN are represented by intern nodes whose left and right children are M and N respectively. For example, $S(KM)(KM)N$ can be represented by the (subterm sharing) graph shown in figure B.1.

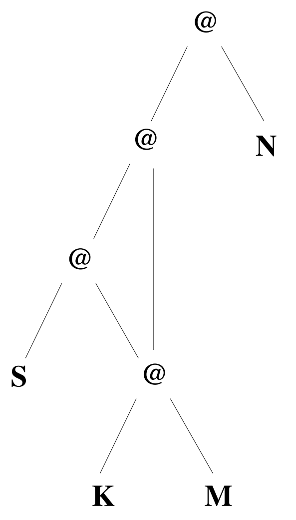

Figure B.1: The combinator graph for $S(KM)(KM)N$

To define the combinator graph reduction rules, we define four auxiliary combinators, S_1, S_2, K_1, and I_1 in addition to S, K, I (henceforth written as S_0, K_0 and I_0). The new combinators are essentially intermediate steps of the S-, K-, and I-reductions, and they serve to localize the transformations such that a node only need to know its immediate parent and children to carry out a reduction. Note that the suffix of each combinator corresponds to the number its child(ren). The reduction rules for the combinators are defined by the schematic shown in figure B.2.

The general strategy of our evaluator is as follows. We define eight kinds of agents, one for each combinator and one for the application. A combinator graph is constructed by setting up a network of agents such that each agent knows its parent and child(ren). Each agent then sends to its parent a private message that includes its type and name(s) of its child(ren), as define by the following agents:

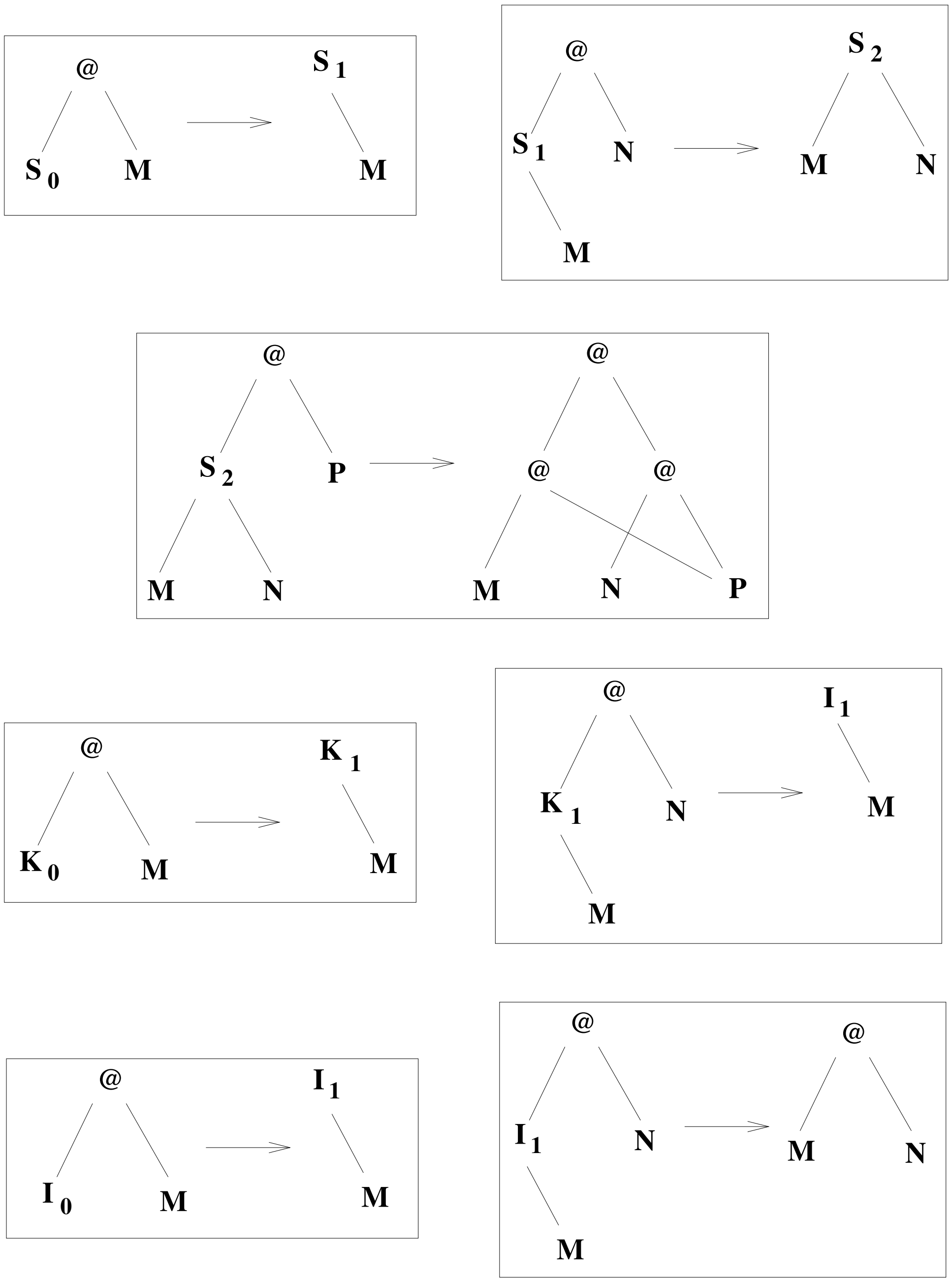

Figure B.2: Combinator Graph Reduction Rules

```
! s0(P) -o w ^ { P(w) * ! w(s0 }
! s1(R,P) -o w ^ { P(w) * ! w(s1,R) }
! s2(L,R,P) -o w ^ { P(w) * ! w(s2,L,R) }
! k0(P) -o w ^ { P(w) * ! w(k0) }
! k1(R,P) -o w ^ { P(w) * ! w(k1,R) }
! i0(P) -o w ^ { P(w) * ! w(i0) }
! i1(R,P) -o w ^ { P(w) * ! w(i1,R) }
```

The actual reductions are carried out by the application agent "@" corresponding to application node. When an application node receives a message from its left child (application is left associative), it carries out one of seven reductions (one for each kind of combinator) according to the message, as defined by the following agent:

```
! @(Left,Right,P) -o
       Left(W) -o W | s0 -o s1(Right,P)
                    & s1,R -o s2(r,right,P)
                    & s2,L,R -o { u,v } ^ { @(u,v,P) *
                                             @(L,Right,u) *
                                             @(R,Right,v) }
                    & k0 -o k1(Right,P)
                    & k1,R -o i1(R,P)
                    & i0 -o i1(Right,P)
                    & i1,R -o @(R,Right,P)
```

To see how the evaluation works, consider the combinator graph in figure B.1. The reduction sequences are represented in figure B.3, where we assume all possible reductions (marked with an *) occur in one step. There are seven reductions but can be completed in 5 steps, when reductions are performed in parallel. The improvement is hardly significant due to the small size of the graph. In a typical graph, there are thousands of nodes and parallel reductions can yield orders of magnitude improvement in running time.

To complete our evaluator, we need another agent to read the outputs of our compiler and constructs the corresponding combinator graph. We will leave out its definition since it is not too illuminating.

It is interesting to note that the evaluator never reduces a common subterm twice if the input graph respects sharing. This can be shown by noticing that only S-reduction introduces common subterms and in the graph reduction rule for S_2, the common subterm is shared.

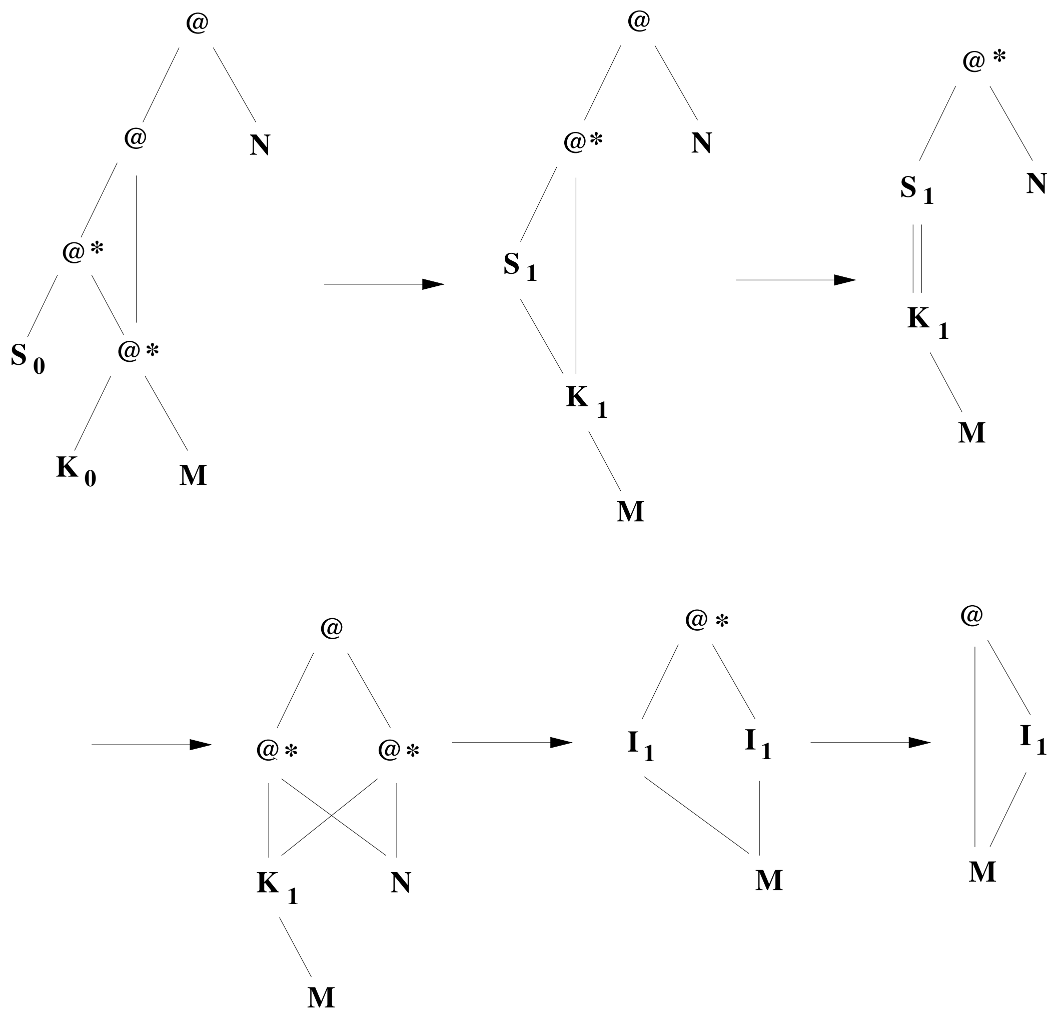

Figure B.3: Parallel graph reduction for $S(KM)(KM)N \xrightarrow{S} KMN(KMN) \xrightarrow{K} M(KMN) \xrightarrow{K} MM$. Nodes marked with an $*$ are reducible and are reduced in the same step. The reduction will take 7 steps if reduced sequentially.

However, the evaluator is not lazy. In fact, it is probably extremely eager. All possible reductions, whether needed or not, will happen. Nonetheless, unnecessary reductions are ignored and will never get into the critical path of the result. In a real system, wasting resources on unnecessary reductions may not be acceptable. We may optimize the evaluator such that a node carries out a reduction only when it knows its result is needed. To do this, each node still broadcasts its identity to the parent and the application agent spawns a new agent that serves to delay the reduction until it is known that the reduction is necessary (similar in principle to the encoding of λ-application in section B.1). For instance, the parent of a K_0 node can safely ignore its right child, and the parent of a S_2 has to delay the reduction of the right child since the reduction may not be necessary.

Appendix C

Abstract Machine Instruction Set

The following tables list the complete set of the abstract machine instructions. The arguments of each instruction are understood as registers and all instructions return results, if any, to the last argument. To avoid verbosity, we use `c`, `msg`, `mth`, `cc`, `lbl` if the register refers to a channel, a message, a method, a condition code, and a label respectively.

C.1 Data Manipulation

Alloc(n, r)	allocate a record of n words
Arity(r, n)	return the arity of r
Set(s, d)	d = s
Store(s, d)	*d = s
Deref(s, d)	d = *s
Cons(car, cdr, cons)	creates a cons-cell
Car(cons, car)	return the first element of a cons-cell
Cdr(cons, cdr)	return the second element of a cons-cell
NewChannel(r)	create a channel
NewScChannel(r)	create a storage channel
NewClosure(n, r)	create a closure for n variables
NewPattern(n, r)	create a pattern for n terms
NewPattern(p1,...,pn, r)	create a pattern using p1, ..., pn
NewMessage(n, r)	create a message with n terms
NewMessage(p1,...,pn, r)	create a message using p1, ..., pn
NewRMethod(p, cl, f, r)	create a reusable method with pattern p, closure cl, body f
NewNMethod(p, cl, f, r)	create a nonce method with pattern p, closure cl, body f
AddMessage(msg, c)	add msg to the messages of c
RemMessage(msg, c)	remove msg from the messages of c
AddMethod(msg, c)	add mth to the methods of c
RemMethod(msg, c)	remove mth from the methods of c
MatchMethod(msg, c, mth)	return the method which matches with msg in c
MatchMessage(mth, c, msg)	return the msg which matches with mth in c

C.2 Communications

Send(msg,c)	send msg to channel c
GcSend(msg,c)	send msg to c, which is a normal channel
RcSend(msg,c)	send msg to c, which is a remote channel
VcSend(msg,c)	send msg to c, which is a virtual channel
ScSend(msg,c)	send msg to c, which is a storage channel
McSend(msg,c)	send msg to c, which is a method channel
Post(mth, c)	post method mth to channel c
RPost(mth, c)	post mth to c, which is a remote channel

C.3 Tests

IsEq(p, q, cc)	return `TRUE` to cc if (p == q), `FALSE` otherwise
IsChannel(c, cc)	return `TRUE` to cc if c is a channel, `FALSE` otherwise
IsSChannel(c, cc)	return `TRUE` to cc if c is a storage channel, `FALSE` otherwise
IsVChannel(c, cc)	return `TRUE` to cc if c is a virtual channel, `FALSE` otherwise
IsMChannel(c, cc)	return `TRUE` to cc if c is a method channel, `FALSE` otherwise
IsRChannel(c, cc)	return `TRUE` to cc if c is a remote channel, `FALSE` otherwise
IsMethod(mth, cc)	return `TRUE` to cc if mth is a method, `FALSE` otherwise
IsRMethod(mth, cc)	return `TRUE` to cc if mth is a reusable method, `FALSE` otherwise
IsNMethod(msg, cc)	return `TRUE` to cc if mth is a nonce method, `FALSE` otherwise
IsMessage(msg, cc)	return `TRUE` to cc if msg is a message, `FALSE` otherwise
IsMessage(msg, cc)	return `TRUE` to cc if msg is a message, `FALSE` otherwise

C.4 Controls

Jeq(p, q, lbl)	jump to lbl if (p == q)
Jcc(cc, lbl)	jump to lbl if (cc == `TRUE`)
Jmp(lbl)	jump to lbl unconditionally
Ret()	terminate

C.5 Remote Interface

Open(host, cc)	open connection to host, return status to cc
Close(host)	close connection to host

C.6 Primitives Agents

The following primitive agents are defined for integer operations. All arguments, i.e. `p1`, `p2`, and `r`, are channels, even though C-like operators are used to described their operations.

Add(p1, p2, r)	r = p1 + p2
Sub(p1, p2, r)	r = p1 - p2
Mul(p1, p2, r)	r = p1 * p2
Div(p1, p2, r)	r = p1 / p2
Mod(p1, p2, r)	r = p1 % p2
Eq(p1, p2, r)	r = (p1 == p2)
Lt(p1, p2, r)	r = (p1 $<$ p2)
Le(p1, p2, r)	r = (p1 $\leq$ p2)
Gt(p1, p2, r)	r = (p1 $>$ p2)
Ge(p1, p2, r)	r = (p1 $\geq$ p2)

Appendix D

Benchmarking Code

The programs used in the obtaining the performance data in chapter 11 are listed below:

- **Lisp**:

```
(defun app (lst1 lst2)
    (if (null lst1)
        lst2
        (cons (car lst1) (app (cdr lst1) lst2))))

(defun nrev (lst)
    (if (null lst)
        '()
        (app (nrev (cdr lst)) (list (car lst)))))
```

- **Prolog**:

```
append([],      Y, Y).
append([A | B], Y, [A | Z]) :- append(B, Y, Z).

nrev([],     []).
nrev([A | B], C) :- nrev(B, D), append(D, [A], C).
```

- **Linear Janus**:

```
! append(X, Y, Z) ⊸
     X | () ⊸ Z $Y
       & (H, T) ⊸ Z(H, append(T, Y))
```

```
! nrev(X, Y) ⊸
     X | () ⊸ Y()
       & (H, T) ⊸ Y(append(nrev(T), ^(H, ^())))
```

D.1 Optimized `append`

Below is the abstract machine instructions generated by the compiler for `append`:

```
Def(append)

  Recv(AP[1], list);
  Jne(list, LJ_NIL, L);
  Fwd(AP[2], AP[3]);
  Ret();

 L:
  Car(list, head);
  Cdr(list, tail);

  NewChannel(tmp)
  Cons(head, tmp, result);

  NewMessage(tail, AP[2], tmp, msg);
  Send(msg, append);

  Send(result, AP[3]);
  Ret();

End
```

From the source code (`R('[H; append(T, L2)]` in particular), it can be seen that `app` does not need to wait for the recursive call to return. If we switch the last two `Send`'s in the code above, the last send would be the recursive to `append`. Since the send is the last instruction in the agent, it is safe to do a jump to the beginning of `append`, instead of a `Send` which has the overhead of creating the process record, enqueueing the record and later dequeuing the record and executing it. The optimized code is as follows:

```
Def(append)

 L_append:
  Recv(AP[1], list);

  Jne(list, LJ_NIL, L);
  Fwd(AP[2], AP[3]);
  Ret();

 L:
  Car(list, head);
  Cdr(list, tail);
  NewChannel(tmp)
  Cons(head, tmp, result);
  NewMessage(tail, AP[2], tmp, msg);
  Send(result, AP[2]);
  Set(msg, AP);
  Jmp(L_append);
  Ret();

End
```

D.2 Optimized `nrev`

For `nrev`, the compiler generates the following instruction sequence:

```
Def(nrev)

  Recv(AP[1], list);
```

```
  Jne(list, LJ_NIL, L)
  Send(LJ_NIL, AP[2]);
  Ret();

 L:
  Car(list, head);
  Cdr(list, tail);

  NewChannel(nrev_tmp);
  NewMessage(tail, nrev_tmp, msg);
  Send(msg, nrev);

  NewChannel(nil);
  Send(LJ_NIL, nil);
  NewChannel(lst_tmp);
  Cons(head, nil, lst);
  Send(lst, lst_tmp);

  NewMessage(nrev_tmp, lst_tmp, AP[2], msg);
  Send(msg, append);
  Ret();

End
```

A number of optimizations are applicable, but the way they are applied is a bit tricky. The first observation is that the last instruction in `nrev` is the call to `append`, which we can inline. Another crucial observation is that a lot of temporary channels are created and used in the recursion. Since the scope of these channels are known, those temporary channels can be optimized. For example, consider the channel `nrev_tmp` which is passed to `append`. After `append` is inlined, the first instruction of `append` would try to fetch a message from `nrev_tmp`. If `nrev_tmp` is a *storage channel* instead, the receive instruction can be eliminated. However, To make `nrev_tmp` a storage channel, there need to be two different entry points in `nrev` in order to distinguish the first call (called with a normal channel) from the recursive calls (called with a storage channel). In addition, instead of calling `append` with three normal channels, three storage channels are used to call `s_append`.

The optimized code is as follows (without the inlining):

```
Def(nrev)

  Recv(AP[1], list);
  Jne(list, LJ_NIL, L)
  Send(LJ_NIL, AP[2]);
  Ret();

 L:
  Car(list, head);
  Cdr(list, tail);

  NewScChannel(nrev_tmp);
  NewMessage(tail, nrev_tmp, msg);
  Send(msg, i_nrev);

  NewScChannel(nil);
  ScSend(LJ_NIL, nil);
  NewScChannel(lst_tmp);
  Cons(head, nil, lst);
  ScSend(lst, lst_tmp);

  NewScChannel(tmp);
  NewMessage(nrev_tmp, lst_tmp, tmp, msg);
  Send(msg, s_append);
  ScRecv(tmp, msg);
  Send(msg, AP[2]);
  Ret();
End
```

Instead `nrev`, the recursive call invokes `i_nrev` which differs from `nrev` in creating the temporary channels as storage channels. Also, instead of calling `append`, `s_append` is called, with all three arguments as storage channels. As mentioned in chapter 9, storage channels avoid deferences through pointers and can often optimized away when the abstract instructions is compiled by a C++ compiler.

```
Def(i_nrev)
  ScRecv(AP[1], list);
```

```
  Jne(list, LJ_NIL, L)
  Send(LJ_NIL, AP[2]);
  Ret();

 L:
  Car(list, head);
  Cdr(list, tail);

  NewScChannel(nrev_tmp);
  NewMessage(tail, nrev_tmp, msg);
  Send(msg, i_nrev);

  NewScChannel(nil);
  ScSend(LJ_NIL, nil);
  NewScChannel(lst_tmp);
  Cons(head, nil, lst);
  ScSend(lst, lst_tmp);

  NewMessage(nrev_tmp, lst_tmp, AP[2], msg);
  Send(msg, s_append);
  Ret();
End
```

Finally, instead of `append`, `s_append` is called, with the storage channels as arguments.

```
Def(s_append)

 L_s_append:
  ScRecv(AP[1], list);
  Jne(list, LJ_NIL, L);
  ScSend(AP[2], AP[3]);
  Ret();

 L:
  Car(list, head);
  Cdr(list, tail);

  NewScChannel(tmp)
  Cons(head, tmp, result);
```

```
  NewMessage(tail, AP[2], tmp, msg);

  ScSend(result, AP[3]);

  Set(AP, msg);
  Jmp(L_s_append);
  Ret();

End
```