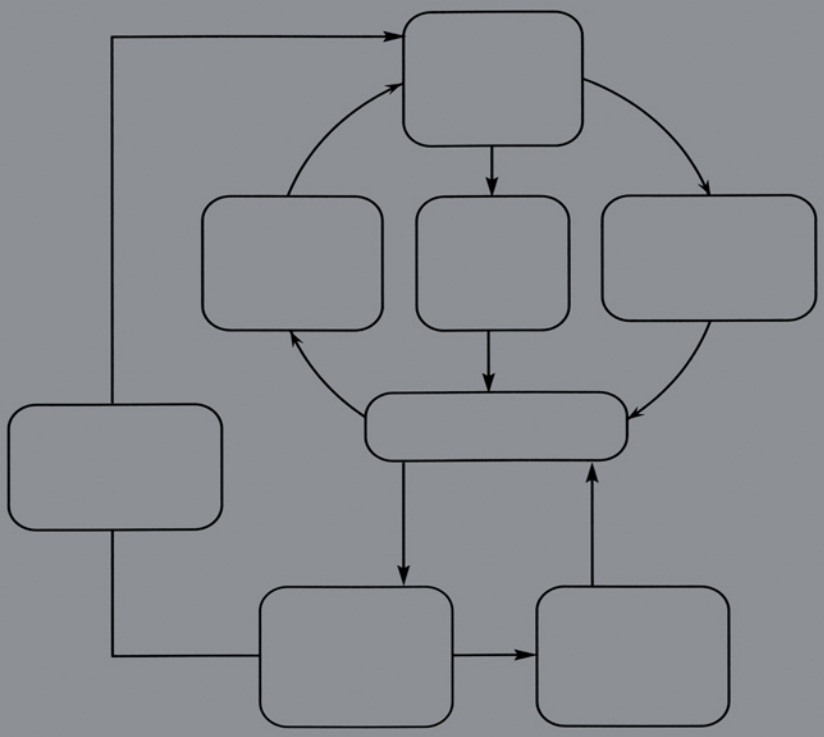
Advanced LISP Technology

Edited by Taiichi Yuasa and Hiroshi G. Okuno



Advanced Lisp Technology

Advanced Information Processing Technology

A series edited by Tadao Saito

Volume 1
Domain Oriented Systems Development: principles and approaches
Edited by K.Itoh, T.Hirota, S.Kumagai and H.Yoshida (1998)

Volume 2
Designing Communications and Collaboration Support Systems
Edited by Y.Matsushita (1999)

Volume 3
Information Networking in Asia
Edited by H.Higaki, Y.Shibata and M.Takizawa (2001)

Volume 4
Advanced Lisp Technology
Edited by T.Yuasa and H.G.Okuno (2002)

Advanced Lisp Technology

Edited by

Taiichi Yuasa
Hiroshi G.Okuno
Graduate School of Informatics
Kyoto University
The Information Processing Society of Japan

London and New York

First published 2002
by Taylor & Francis
11 New Fetter Lane, London EC4P 4EE

Simultaneously published in the USA and Canada
by Taylor & Francis Inc,
29 West 35th Street, New York, NY 10001

Taylor & Francis is an imprint of the Taylor & Francis Group

This edition published in the Taylor & Francis e-Library, 2005.

British Library Cataloguing in Publication Data
A catalogue record for this book is
available from the British Library

Library of Congress Cataloguing in Publication Data
A catalog record for this book has been requested

ISBN 0-203-30087-4 Master e-book ISBN

ISBN 0-203-34550-9 (Adobe eReader Format)
ISBN 0-415-29819-9 (Print Edition)

Contents

Series Foreword

The Information Processing Society of Japan (IPSJ) is the top academic institution in the information processing field in Japan. It has about thirty thousand members and promotes a variety of research and development activities covering all aspects of information processing and computer science.

One of the major activities of the society is publication of its transactions containing papers covering all the fields of information processing, including fundamentals, software, hardware, and applications. Some of the papers are published in English, but because the majority are in Japanese, the transactions are not suitable for non-Japanese wishing to access advanced information technology in Japan. IPSJ therefore decided to publish a book series titled "Advanced Information Technology in Japan."

The series consists of independent books, each including a collection of top quality papers, from mainly Japanese sources of a selected area in information technology. The book titles were chosen by the International Publication Committee of IPSJ so that they enable easy access to information technology from international readers. Each book contains original papers and/or those updated or translated from original papers appearing in the transactions or internationally qualified meetings. Survey papers to aid understanding of the technology in Japan in a particular area are also included.

As the chairman of the International Publication Committee of IPSJ, I sincerely hope that the books in the series will improve communication between Japanese and non-Japanese specialists for their mutual benefit.

Tadao Saito
Series Editor
Chairman
International Publication Committee
the Information Processing Society of Japan

Preface

Research and development activities related to Lisp have been quite active in Japan. Many Lisp languages and many Lisp processors have been designed and developed here.

There were three booms in Lisp research; in the early seventies, in the mid-eighties, and in the late-eighties. During the first boom, some experimental Lisp systems were developed, and some also came to be practically used. At that time, available computing power and memory space were small, as was the Lisp community. The second boom was triggered by the explosion of interest in AI, which was partly due to the Fifth Generation Computer System Project in Japan. Although the FGCS project advocated Prolog as a primary AI language, Lisp also shared the attention paid to AI research.

The third boom was caused by the emergence of Common Lisp. Available computing power and memory space increased greatly, and the Lisp community also became larger. Many Lisp systems were developed at this time. In particular, KCL (Kyoto Common Lisp) has been used not only in Japan but also in the world. Major research and development activities up to 1990 were reported in the special section on "Lisp systems in Japan" of *Journal of Information Processing* (vol.13, no.3, 1990).

Recently many parallel and distributed processors are available both experimentally and commercially, which prompts academia and industry to do research in parallel and distributed processing. This book focuses on recent activities, including parallel and distributed Lisp systems, memory managements for those systems and other related topics.

The papers in this book are organized into groups by subject:

The papers in Part I report contributions to Lisp system design and implementation on a wide variety of parallel and distributed computing environments, such as SMPs, distributed-memory parallel machines, SIMD machines, and network-connected workstations. These systems attempt various strategies to provide the base system with constructs for parallel computation, from low-level communication primitives to high-level semi-automatic parallelizing constructs.

Most language features proposed in the papers in Part II are related with parallel and distributed Lisp processors. Included features are: evaluation strategy for parallel symbolic computation, extension of first-class continuations for parallel Scheme systems, and light-weight process for real-time symbolic computation. The last paper in Part II proposes restricted but efficient first-class continuations.

Part III of this book consists of papers on memory management and garbage collection. Topics in these Papers are: parallel garbage collection for parallel Lisp systems, hybrid incremental garbage collection for parallel Lisp systems, garbage collection for Lisp systems on SIMD architectures, and stop-and-collect type of garbage collection with sliding rather than copying.

Papers in Part IV are related with programming environment. The first paper handles system's management of Lisp macros. The second paper proposes user interface for Lisp printers.

As will be clear from the collection of papers in this book, the main interest of Lisp research in Japan has been on parallel and distributed processing. The current movement is to develop new application areas, which requires high productivity, dynamic features, and efficiency of Lisp in the information network society, though it is too early to include papers along this movement in this book.

Taiichi Yuasa
Hiroshi G.Okuno
Kyoto and Tokyo
At the beginning of the 21st Century

The Authors

Yoshiji Amagai received his B.E. and M.E. degrees in computer science and communication engineering from the University of Electro-Communications in 1983, 1985. He has been in Nippon Telegraph and Telephone Corporation from 1985, and is currently working for NTT Network Innovation Laboratories.

Akihiro Fushimi was born in 1971. He received his M.E. degree from Toyohashi University, of Technology, in 1995. He has been working in NTN Corp. since 1995 and now is a system engineer of Advanced Operation Development Dept. of NTN. His current interests are network architecture and programming language implementation.

Katsumi Hatanaka was born in 1969. He received his M.E. degree from Toyohashi University, of Technology, in 1994. He has been working in OKI Electric Industry Co., Ltd. since 1994 as a database engineer. His current interests are data warehouse and knowledge discovery in databases. He is a member of IPSJ.

Mitsuru Ishida was born in 1955. He received the degree of M.E. degree in Education from Tokyo Gakugei University, Tokyo, Japan, in 1982. Since 1982, he has been a teacher of high school. He also received the B.E. degree in computer science from the University of Electro-Communications in 1992. He is now a doctoral student in computer science at the University of Electro-Communications. His current research interests include storage management of both concurrent and real-time types.

Takayasu Ito received his B.E. in electrical engineering in 1962 from Kyoto University, his M.S. in computer science in 1967 from Stanford University, and his Dr. Eng. degree in 1970 from Kyoto University. In 1962–1977 he was with Central Research Laboratory of Mitsubishi Electric Corp with on leave to AI Laboratory of Stanford University from 1965 until 1968. Since 1978 he has been a Professor at Tohoku University, and now is a Professor, Department of Computer and Mathematical Sciences, Graduate School of Informations Sciences. Since 1962 his research interests are in AI and pattern recognition, Lisp and functional languages, theories of programs and semantics. His current interests are in extensions of PaiLisp and ISLisp that he advocated, constructive theories of concurrency and execution, and design of new internet languages. He is Associate Editor of Higher-Order and Symbolic Computation, and a member of Editorial board of Information and Computation. Also, he served conference chairs of TACS'91, TACS'94, TACS'97, and IFIP TCS2000, and he is the conference chair of TACS 2001. He is one of the first IPSJ Fellows.

Teruo Iwai received his B.E. in 1988, his M.E. degrees in 1990 both in mathematics from Keio University. He is a research fellow at Keio University. His current research interests are Lisp systems and parallel processing.

Shin-ichi Kawamoto received his B.E. and M.E. in information engineering in 1991 and 1993, respectively, and his Dr. degree in information science in 1998, all from Tohoku University. He was a research associate from 1996 until 1998, working on the PaiLisp system. Since 1998 he has been with Central Research Laboratory of Hitachi Ltd., where he works on basic software systems for high-perfeormance computing architectures.

Hideki Koike received his B.E. in mechanical engineering in 1986 and his M.E. and Dr. Eng. degrees in information engineering in 1988 and 1999, respectively, all from the University of Tokyo. Currently he is an Associate Professor of Graduate School of Information Systems in University of Electro-Communications, Tokyo.

Tsuneyasu Komiya received his B.E., M.E. and Dr. Eng. degrees from Toyohashi University of Technology in 1991, 1993 and 1996, respectively. He is currently an instructor in the Graduate School of Informatics, Kyoto University. His research interests include symbolic processing languages and parallel languages. He received an IPSJ best paper award in 1997. He is a member of IPSJ.

Atusi Maeda received his B.E. in 1986, his M.E. in 1988 and his Ph.D. (Eng.) degrees in 1997 all in mathematics from Keio University. He was a Research Associate at University of Electro-Communications from 1997 to 1999. He is currently an Assistant Professor at Institute of Information Sciences and Electronics, University of Tsukuba. His research interests are in design and implementation of programming languages, compilers and parallel programming.

Shogo Matsui received his B.E. in 1982, his M.E. in 1984 and his Ph.D. (Eng.) degree in 1991 all in mathematical engineering from Keio University. He joined the faculty at Kanagawa University in 1989, and now is an Associate Professor in the Department of Information Science. His research interests are in Lisp systems including Lisp machines and parallel garbage collection.

Toshihiro Matsui received his B.E. in instrumentation engineering 1980, his M.E. in information engineering in 1982, and Dr. Eng. degrees in information engineering in 1991, all from the University of Tokyo. He joined Electrotechnical Laboratory in 1982, and conducted research on programming systems for robotics. He was a visiting scholar of Stanford University, MIT AI Lab., and Australian National University in 1991, 94 and 99. He received Research/Best-Paper Awards from International Symposium of Robotics Research (ISRR) in 1989, Robotics Society of Japan in 1989, Moto-oka Memorial in 1992, etc. He is now Senior Research Scientist of ETL and leading a research group for a mobile office robot project.

Yoshitaka Nagano was born in 1967. He received his M.E. degree from Toyohashi University, of Technology, in 1993. He has been working in NTN Corp. since 1993 and now is a system engineer of R&D Dept. of NTN. His current interest is computer vision for industrial automation.

Masakazu Nakanishi received his B.E. in 1966, his M.E. in 1968 and his Dr. Eng. all in administrative engineering from Keio University. He joined the faculty at Keio University in 1969 and now is a Professor in Department of Information and Computer Science. He died in 2000. He developed the first Lisp system in Japan. His main research interests lie in languages for symbol processing and artificial intelligence and computer education.

Hiroshi Nitta was born in Kanagawa, Japan on November 15, 1971. He received the B.E. degree in systems engineering from Ibaraki University, Ibaraki, Japan, in 1995. He received the M.E. degree in computer science from the University of Electro-Communications, Tokyo, Japan, in 1997. He is now a doctoral student in computer science at the University of Electro-Communications. His current research interests include storage management.

Hiroshi G.Okuno received his B.A. in Pure and Applied Sciences in 1972, and his Ph.D in Computer Science in 1996, both from the University of Tokyo. He worked for Nippon Telegraph and Telephone Corporation for more than 26 years, and has joined Kitano Symbiotic Systems Project, Japan Science and Technology Corporation in 1998. From 1999 to 2001, he was a Professor of Information Sciences, Science University of Tokyo. He is currently a Professor at the Graduate School of Informatics, Kyoto University.

Toshiharu Ono received his B.E. in Mathematics in 1950 from the University of Rikkyou and his M.E. degree in Information Engineering in 1990 from the University of Yamanashi. He is instructor of Electrical and Computer at Nishi Tokyo Science University. His research is in distributed systems, and computer languages. Since 1971 he has worked as an engineer at "NEC" and has been involved in the design and implementation of the Fortran compiler.

Satoshi Sekiguchi received his B.S. degree in information science from the University of Tokyo in 1982 and the M.E. degree in information engineering from the University of Tsukuba in 1984 respectively. He joined with Electrotechnical Laboratory in 1984, and conducted research on parallel computing systems for scientific computing. He was a visit scholar of National Center for Atmospheric Research in 1996, and a visiting scientist in ETH Zurich in 1998. He received Ichimura Academic Award in 1989, Best-Paper Award from Information Processing Society Japan (IPSJ) in 1995, etc. He is now Senior Research Scientist of ETL and leading a research group on the high performance computing group. His research interests include parallel algorithms for scientific computations, global computing and cluster computing systems.

Satoko Takahashi received her B.E. degree in mathematics in 1994 and her M.E. degree in computer science in 1996 both from Keio University. She works at Media Technology Development Center, NTT Communications Corporation Her research interests include development of CTI systems.

Ikuo Takeuchi received his B.S. degrees in mathematics in 1969 and his M.S. degrees in mathematics in 1971, all from the University of Tokyo. He had been working for Nippon Telegraph and Telephone Corporation since 1971 till 1997, and now is a Professor of the University of Electro-Communications.

Tetsurou Tanaka received his B.E. in mathematical engineering in 1987 and his M.E. and Dr. Eng. degrees in information engineering in 1989 and 1992, respectively, all from the University of Tokyo. He is now an Associate Professor at Information Technology Center, the University of Tokyo.

Yoshio Tanaka received his B.E. in 1987, his M.E. in 1989 and his Ph.D. (Eng.) degree in 1995 all in mathematics from Keio University. He was working at Real World Computing Partnership from 1996 to 1999. He is currently the team leader of Grid Infraware Team at Grid Technology Research Center, National Institute of Advanced Industrial Science and Technology. His current research interests include Grid computing, performance evaluation of parallel systems, and memory management.

Motoaki Terashima was born in Shizuoka, Japan in June 8, 1948. He received the degrees of B.Sc. (1973), M.Sc. (1975) and D.Sc. (1978) in Physics from the University of Tokyo for his work on computer science. Since 1978, he has been a research associate of the Department of Computer Science, University of Electro-Communications, and is currently an Associate Professor of Graduate School of Information Systems. He was a visiting scholar at the Computer Laboratory of the Cambridge University in 1992. His current research interests include programming language design and implementations, memory management, and symbolic and algebraic manipulation systems. He is a member of IPSJ, ACM and AAAI.

Masayoshi Umehara received his B.E. and M.E. in information engineering in 1994 and 1996, respectively, from Tohoku University. He worked on a PaiLisp complier for his M.E. thesis. Since 1996 he has been with NEC Communications Division.

Eiiti Wada received his B.S. degree from the University of Tokyo. Then he joined Professor Hidetosi Takahasi's Laboratory of the Physics Department, University of Tokyo, as a graduate student. In 1958, when the parametron computer, PC-1 was completed, he spent many hours for preparing software library. In

1964, he moved to the Faculty of Engineering of the University of Tokyo, as an Associate Professor. From 1973 he spent one year at MIT, Project MAC. In 1990's, with his graduate students, he worked for developing the Chinese character fonts generation system, in which character shapes were composed from the definition of the elementary parts based on the synthesis algorithms. He has been the member of the IFIP WG2.1. Presently he is a Professor Emeritus of the University of Tokyo and the Executive Advisor of Fujitsu Laboratories.

Haruaki Yamazaki received the B.S. degree in Mathematics in 1970 from Nagoya University, Japan. He subsequently joined OKI electric Industry Co., Ltd and engaged in research and development on distributed system. He received the M.S. degree in computer science in 1977, from University of Illinois at Urbana-Champaign. He received his doctorial degree in Engineering from Nagoya University in 1986. Since 1992, he has been a Professor at Department of Computer Science, Yamanashi university, kofu, Japan. His current interests are in distributed systems, distributed ar tificial intelligence. He is a member of Institution of Electronics and Communication Engineers of Japan, Information Processing Society of Japan, Artificial Intelligence Society of Japan and American Association for Artificial Intelligence.

Kenichi Yamazaki received his B.E. degrees in communication engineering in 1984 and his M.E. degrees in information engineering in 1986, all from Tohoku University, and received his Ph.D. degrees in 2001 from the University of Electro-Communications. He had been in Nippon Telegraph and Telephone Corporation from 1986, and is currently working for NTT DoCoMo Network Laboratories.

Taichi Yasumoto was born in 1966. He received his M.E. degree from Toyohashi University of Technology in 1990. He is currently an Associate Professor at Department of Mathematical Science, Aichi University of Education. His research interests are symbolic processing languages and parallel processing. He is a member of IPSJ, IEICE and JSSST.

Masaharu Yoshida received his B.E. degrees in electrical engineering in 1976 and his M.E. degrees in electrical engineering in 1978, all from Chiba University. He has been in Nippon Telegraph and Telephone Corporation from 1978, and is currently working for NTT-IT CORPORATION.

Taiichi Yuasa was born in Kobe, in 1952. He received the Bachelor of Mathematics degree in 1977, the Master of Mathematical Sciences degree in 1979, and the Doctor of Science degree in 1987, all from Kyoto University. He joined the faculty of the Research Institute for Mathematical Sciences, Kyoto University, in 1982. He is currently a Professor at the Graduate School of Informatics, Kyoto University. His current area of interest include symbolic computation, programming language systems, and massively parallel computation. He received an IPSJ best paper award in 1997. Dr. Yuasa is a member of ACM, IEEE, Information Processing Society of Japan, the Institute of Electronics, Information and Communication Engineers, and Japan Society for Software Science and Technology.

1
A Multi-Threaded Implementation of PaiLisp Interpreter and Compiler Using the Steal-Help Evaluation Strategy

Takayasu Ito

Shin-ichi Kawamoto

Masayoshi Umehara

Department of Computer and Mathematical Sciences

Graduate School of Information Sciences

Tohoku University, Sendai, Japan

ABSTRACT

PaiLisp is a parallel Lisp language with a rich set of concurrency constructs like **pcall**, **pbegin**, **plet**, **pletrec**, **par-and**, **pcond** and **future**, and an extended **call/cc** is introduced to support *P*-continuation. The Eager Task Creation (ETC, for short) is a commonly-used implementation strategy for concurrency constructs. But ETC incurs excessive process creation which causes degradation of performance in parallel evaluation of a program parallelized using concurrency constructs. The Steal-Help Evaluation (SHE, for short) is introduced as an efficient parallel evaluation strategy which enables to suppress excessive process creation. This paper explains a multi-threaded implementation of PaiLisp interpreter and compiler, in which the ETC and SHE strategies for PaiLisp concurrency constructs are realized. Some experimental results of using a set of benchmark programs show that the SHE strategy is more efficient than the ETC strategy.

1
Introduction

There are several parallel Scheme languages like Multilisp[3]) and PaiLisp[8]), designed for shared memory architectures. Multilisp may be considered to be a minimal extension of Scheme into concurrency, introducing an interesting concurrency construct **future**. PaiLisp is an extension of Scheme with a rich set of concurrency constructs like **pcall**, **pbegin**, **plet**, **pletrec**, **pif**, **par-and**, **par-or**, **pcond** and **future**. PaiLisp may be considered to be a superset of Multilisp as a parallel Scheme language. A PaiLisp interpreter was implemented using the *Eager Task Creation* (ETC, for short) as reported in the reference 10). ETC is a commonly-used technique in implementing various parallel programming languages, but it usually incurs a serious problem of excessive process creation, in particular, in evaluating a recursively-defined parallel program. Under the ETC strategy it often occurs that for a given sequential program its parallelized version runs slower than the original sequential one. This is because when a concurrency construct is encountered the ETC strategy creates processes specifiied by it without any concern of availability of processors. The *Steal-Help Evaluation* (SHE, for short) is proposed as an efficient evaluation strategy of concurrency constructs in parallel Scheme systems[15, 16]. SHE enables to suppress excessive process creation, since it creates processes only when an idle processor is available in evaluating an

$$
\begin{array}{lll}
D & ::= & \textbf{(define } x\ E\textbf{)} \ \mid \textbf{(define } f\ \textbf{(lambda } (x_1 \cdots x_n)\ E_1 \cdots E_m\textbf{))} \\
E & ::= & V \\
 & \mid & \textbf{(}F\ E_1 \cdots E_n\textbf{)} \mid \textbf{(begin } E_1 \cdots E_n\textbf{)} \\
 & \mid & \textbf{(if } E_1\ E_2\ E_3\textbf{)} \mid \textbf{(cond } (E_{11}\ E_{12}) \cdots (E_{n1}\ E_{n2})\textbf{)} \\
 & \mid & \textbf{(let } ((x_1\ E_1) \cdots (x_m\ E_m))\ E_{m+1} \cdots E_n\textbf{)} \\
 & \mid & \textbf{(let* } ((x_1\ E_1) \cdots (x_m\ E_m))\ E_{m+1} \cdots E_n\textbf{)} \\
 & \mid & \textbf{(letrec } ((x_1\ E_1) \cdots (x_m\ E_m))\ E_{m+1} \cdots E_n\textbf{)} \\
 & \mid & \textbf{(and } E_1 \cdots E_n\textbf{)} \mid \textbf{(or } E_1 \cdots E_n\textbf{)} \\
 & \mid & \textbf{(set! } x\ E\textbf{)} \mid \textbf{(call/cc } E\textbf{)} \mid \textbf{(call/ep } E\textbf{)} \\
 & \mid & \textbf{(delay } E\textbf{)} \mid \textbf{(force } E\textbf{)} \\
 & \mid & \textbf{(pcall } F\ E_1 \cdots E_n\textbf{)} \mid \textbf{(par } E_1 \cdots E_n\textbf{)} \mid \textbf{(pbegin } E_1 \cdots E_n\textbf{)} \\
 & \mid & \textbf{(plet } ((x_1\ E_1) \cdots (x_m\ E_m))\ E_{m+1} \cdots E_n\textbf{)} \\
 & \mid & \textbf{(pletrec } ((x_1\ E_1) \cdots (x_m\ E_m))\ E_{m+1} \cdots E_n\textbf{)} \\
 & \mid & \textbf{(pif } E_1\ E_2\ E_3\textbf{)} \\
 & \mid & \textbf{(pcond } (E_{11}\ E_{12}) \cdots (E_{n1}\ E_{n2})\textbf{)} \mid \textbf{(pcond\# } (E_{11}\ E_{12}) \cdots (E_{n1}\ E_{n2})\textbf{)} \\
 & \mid & \textbf{(par-and } E_1 \cdots E_n\textbf{)} \mid \textbf{(par-or } E_1 \cdots E_n\textbf{)} \\
 & \mid & \textbf{(future } E\textbf{)} \mid \textbf{(call/pcc } E\textbf{)} \mid \textbf{(call/pep } E\textbf{)} \\
 & & \\
V & ::= & \textit{Constant} \mid \textit{Variable} \\
 & \mid & \textbf{(lambda } (x_1 \cdots x_n)\ E_1 \cdots E_m\textbf{)} \mid \textbf{(exlambda } (x_1 \cdots x_n)\ E_1 \cdots E_m\textbf{)} \\
 & & \\
F & ::= & \textit{Primitive-function} \mid \textit{User-defined-function} \mid \textit{System-function} \\
 & \mid & \textbf{(lambda } (x_1 \cdots x_n)\ E_1 \cdots E_m\textbf{)}
\end{array}
$$

Figure 1: Syntax of PaiLisp

expression annotated by a concurrency construct. A multi-threaded implementation of PaiLisp interpreter and compiler using the SHE strategy is implemented on a DEC7000 with six Alpha processors under OSF/1 OS. This paper explains a multi-threaded implementation of PaiLisp interpreter and compiler, called PaiLisp/MT, in which the ETC and SHE strategies are realized. Some experimental results of PaiLisp/MT are also given, using a set of benchmark programs, and they show that the SHE strategy is actually effective and efficient as a parallel evaluation strategy.

2
PaiLisp

PaiLisp is designed as a parallel Scheme language with a rich set of concurrency constructs as follows.

The meanings of sequential constructs are same with those of Scheme[1)], and the meanings of concurrency constructs are given so as to preserve meanings of their corresponding sequential ones[15, 16)]. For example, the meanings of **pcall**, **pbegin**, **plet**, **pif**, **par-and** and **future** are as follows.

(pcall *f* $e_1 ... e_n$): After evaluating e_1, …, e_n in parallel, ***f*** is evaluated, and its resultant value is applied to the values of e_1, ..., e_n.

(pbegin $e_1 \ldots e_n$**):** e_1, …, e_n are evaluated in parallel, and after their termination the value of e_n are returned as the value of this **pbegin** expression.

(plet $((x_1\ e_1) \ldots (x_n\ e_n))\ E_1 ... E_m$**):** After evaluating e_1, …, e_n in parallel and binding their values to the corresponding variables, the environment is extended with these bindings. Then under the extended environment E_1, …, E_m are evaluated in parallel, and after their termination the value of E_m is returned as the value of this **plet** expression.

(pif $e_1\ e_2\ e_3$**):** The expressions e_1, e_2 and e_3 are evaluated in parallel, and when the value of e_1 becomes *true* the value of e_2 is returned, killing (that is, forcing termination of) evaluation of e_3, and when the value of e_1 becomes *false* the value of e_3 is returned, killing evaluation of e_2.

(par-and $e_1 ... e_n$**):** e_1, …, e_n are evaluated in parallel, and if one of them yields *false* then *false* is returned, killing all the remaining processes created in evaluating e_1, …, e_n. If none of e_1, …, e_n yields *false* then the value of e_n is returned.

(future e**):** When this expression is encountered the future-value for e is returned, and a new child process of evaluating e is created. The evaluation on the parent process is continued, using the future-value for e. When the true value of the future-value for e is required, the **force** operation is performed to obtain it.

PaiLisp is featured in introducing a parallel construct corresponding to each sequential Scheme construct, and in addition, PaiLisp-Kernel is extracted. PaiLisp-Kernel=Scheme+**{spawn, suspend, exlambda, call/pcc}** is a small subset of PaiLisp, in which the meanings of other PaiLisp constructs can be expressed[8].

3
Outline of PaiLisp Interpreter and Compiler

In this section we outline PaiLisp/MT, a multi-threaded implementation of PaiLisp interpreter and compiler realized on a DEC7000 with six Alpha processors using the P-thread library of OSF/1 OS. PaiLisp/MT is implemented using the *Register Machine* (RM) model of Abelson and Sussman[1], and the ETC and SHE strategies are used in evaluating concurrency constructs of PaiLisp. The PaiLisp interpreter is a parallel evaluator which consists of six sequential Scheme interpreters under a shared memory, and the PaiLisp compiler is a compiler to translate a PaiLisp program into the corresponding concurrent RM program, which is eventually translated into the corresponding C program. The resultant compiled C program is executed under the run-time mechanism equipped to PaiLisp/MT. Thus, PaiLisp/MT consists of the following.

(1) PaiLisp/MT is implemented on a DEC7000 with six Alpha processors using the P-thread library of OSF/1 OS.
(2) PaiLisp/MT is realized, using the RM model, in which an evaluator **(Evaluator)** resides, and each RM is realized as a thread of P-thread library of OSF/1 OS. Note that the RM used in PaiLisp/MT is extended to include several registers and their related commands to handle concurrency constructs.
(3) PaiLisp/MT is organized as Figure 2 (a), and its interpreter is a parallel Scheme interpreter, whose evaluator **(Evaluator)** resides in each RM. The **Evaluator** consists of five modules as shown in Figure 2 (b).
(4) In the **Evaluator** of PaiLisp/MT three parallel evaluation strategies are implemented in the **parallel construct** module to support the ETC, SHE and LTC strategies. Note that the LTC submodule is applicable only to an expression annotated by **future.**
(5) The SST (Stealable Stack) is introduced to implement and support the SHE strategy; that is, the SST is equipped into each RM as shown in Figure 2 (a).
(6) The PaiLisp/MT compiler is designed to be a compiler linked to the PaiLisp interpreter to support its run-time routine; that is, compiled codes of a PaiLisp program are executed, linking to the PaiLisp/MT interpreter.
(7) The PaiLisp compiler translates a PaiLisp program into a RM program, which will be eventually translated into the corresponding C program for evalation.
(8) Parallel evaluation strategies equipped in PaiLisp/MT are ETC and SHE, and in addition the LTC (Lazy Task Creation) strategy is realized for the **future** construct to compare the SHE-based **future** and the LTC-based **future.**

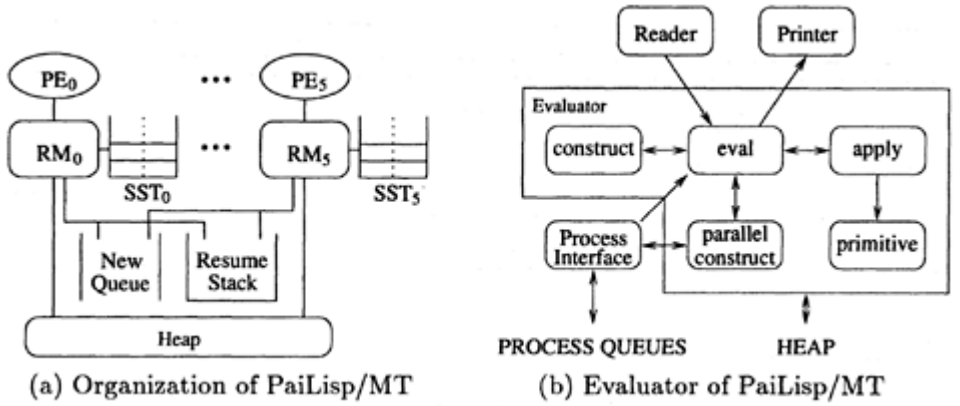


Figure 2: Organization of PaiLisp/MT

3.1
Register Machine Model and Behaviors of PaiLisp/MT

Each RM is an extended version of the Abelson-Sussman Register Machine[1)], consisting of registers, stacks and an evaluator, as shown in Figure 2. In addition to the val, fun, env, argl, cont, exp and unev registers, two registers for the compiler are introduced; the process register, which is an internal register to keep record of the current process-id, and the argcount register to record the number of argument expressions. The environment in each RM will be updated in *deep-binding*. The set of RM commands is given in Appendix; they are extended from that of Abelson-Sussman's RM model so as to accommodate PaiLisp's facilities in compilation.

Under the ETC strategy the PaiLisp/MT interpreter works as follows. Six RMs will be executed in parallel, running each RM on the corresponding processing element (PE). When an expression containing a concurrency construct is evaluated new concurrent processes will be created. The newly created processes will be stored into the New Queue. If an idle processor is available a process in the New Queue will be executed. When a suspended process is resumed for execution, it will be stored into the Resume Stack. An idle RM checks the Resume Stack, and if there is any PaiLisp process the RM gets a process from the Resume Stack for evaluation. If there is no process in the Resume Stack then the RM checks the content of the New Queue, and if there exist any PaiLisp processes the RM gets a process from the New Queue for evaluation. The RMs share a heap area for their evaluation. In the case of the SHE strategy the SST will be used instead of the New Queue in process creation, as is explained below.

The PaiLisp/MT compiler translates a source PaiLisp program into the corresponding RM program, which will be translated into the C program. The compiled object code of the resultant C program will be executed under the run-time routines equipped to the PaiLisp/MT interpreter. The run-time routines consist of the codes for primitive functions, function application, process management and code-loading.

3.2
Evaluator of the PaiLisp/MT interpreter

The **Evaluator** that resides in each RM is a Scheme interpreter, consisting of five modules; **eval, apply, primitive, construct,** and **parallel construct.** Note that the **parallel construct** module consists of three

modules, the ETC, SHE and LTC modules to realize parallel evaluation of PaiLisp expressions. Among six RMs there is one RM, called the *Master Processor,* equipped with **Reader** and **Printer.** Other processors, called the *Slave Processor,* are those obtained from the *Master Processor,* removing **Reader** and **Printer.**

The PaiLisp/MT interpreter is realized as a system with multiple evaluation strategies; the sequential evaluation, ETC and SHE. Similarly, the PaiLisp/MT compiler is realized as a system with the multiple evaluation strategies.

4
The Steal-Help Evaluation Strategy and Its Implementation in the PaiLisp/MT Interpreter

The Steal-Help Evaluation (SHE) strategy[15, 16] is an efficient parallel evaluation strategy that can suppress excessive process creation. In the SHE strategy a processor called the home processor starts evaluating a parallel expression in a sequential evaluation mode, obtaining an expression from the top of the Stealable Stack (SST), while an idle processor steals and evaluates an expression, obtaining it from the bottom of the SST. The SHE strategy for (**pcall** $f\ e_1 \ldots e_n$) works as follows.

A processor called the home processor that invoked (**pcall** $f\ e_1 \ldots e_n$) starts evaluating e_1, and after evaluating e_1 the home processor proceeds to evaluating an expression obtaining a pair of an expression and its shared object from the top of the SST sequentially, while an idle processor called the helper steals a pair of an expression and its shared object from the bottom of the SST, and the stolen expression will be evaluated under the environment contained in the shared object. Note that when a parallel expression like (**pcall** $f\ e_1 \ldots e_n$) is nested in another parallel expression it may be stolen by an idle processor, which will become a home processor of the stolen expression.

The SHE strategy for a parallel expression consists of the sequential evaluation mode and the steal-help evaluation mode[15, 16]. Before explaining these two modes we explain about the stealable stack and the shared object, required in correct implementation of the SHE strategy.

Stealable Stack

The *Stealable Stack* (SST, for short) with the pointers *top* and *bottom* is equipped in each RM, and a slot of the SST is formed from a pointer to an expression and a pointer to a shared object. An idle processor (a helper) will obtain a pair of an expression and a shared object from the bottom of the SST of the home processor. The helper's action of obtaining the pair is called a stealing action.

Shared Object

A *shared object* (SO, for short) is necessary for correct evaluation of an expression stolen from the SST by a helper. A shared object is formed from (a pointer to) the environment, together the name of the current concurrency construct, the name of the process that invoked the current parallel expression, the counter of counting the number of unevaluated expressions, and a mutex variable of a thread for exclusive access to the counter.

4.1
SHE-based Implementation of pcall

The SHE strategy consists of two evaluation modes, the sequential evaluation mode and the steal-help evaluation mode. We explain here how the SHE strategy for **pcall** is implemented in the PaiLisp/MT interpreter.

(1) Sequential evaluation mode for (pcall *f* $e_1 \ldots e_n$)

When RM_i encounters (**pcall** *f* $e_1 \ldots e_n$) for evaluation, the RM_i becomes the home processor for this **pcall** expression, and its evaluation is performed in the sequential evaluation mode as follows.

1: create a shared object SO;
2: store the expressions e_2, …, e_n together with their shared object SO into the SST_i in the order of placing (e_n . SO) at the bottom towards (e_2 . SO) at the top;
3: evaluate e_1 with the SO on the home processor, and store the resultant value into the first place of the arg list;
4: counter :=counter–1;
5: while (counter > 0)
6: if (SST_i is not empty)
7: then get an argument expression from the SST_i pointed by top_i;
evaluate it;
save the resultant value into the corresponding arg list;
counter:=counter–1;
8: else wait until completion of handling all the argument expressions;
9: evaluate *f*, and apply its resultant value to the values of the argument expressions stored in the arg list;
10: return the resultant value to the val register of RM_i.

(2) Steal-help evaluation mode for (pcall *f* $e_1 \ldots e_n$)

A processor RM_j would become idle after completion of evaluating expressions, and also it would become idle by suspension of evaluation. Such an idle processor RM_j can perform the following actions in evaluating an expression.

(a) if there exist any processes in the Resume Stack (RS) then a process in the RS is evaluated on the RM_j until no process becomes available from the RS; otherwise, the following *steal-help* evaluation mode is performed.
(b) the idle processor RM_j will find a processor RM_i whose SST_j is not empty, and the RM_j performs the following actions in the steal-help evaluation mode for the SST_i.

1: RM_j will steal an expression and its shared object SO from the SST_i in RM_i;
2: RM_j creates the process object for evaluating the expression stolen from SST_i;
3: RM_j will evaluate the stolen expression, and the resultant value will be saved into the corresponding place of the arg list in the RM_j;
4: counter :=counter–1;
5: if (counter=0)
6: then resume the evaluation on RM_i;
7: terminate the process of evaluating the stolen expression;

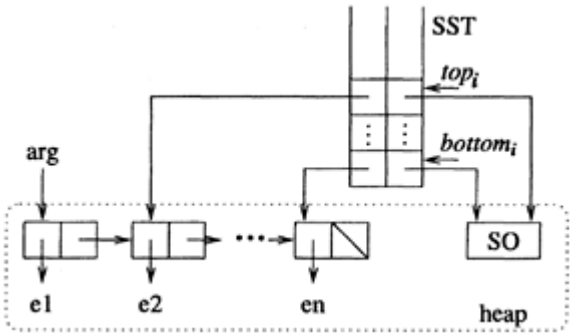


Figure 3: SST and its pointer organization

The SST and its pointer organization is shown in Figure 3. The scheduling strategy in stealing an expression from the RMs is the one like Round-Robin scheduling, designed so as to reduce access competition among the RMs.

4.2
On SHE-based implementations of other concurrency constructs

Other PaiLisp concurrency constructs can be implemented, following the method explained in the references 6) and 7), but some implementational considerations must be taken into account as in the case of **pcall.** In this paragraph we give only some short comments on such implementations.

(1)
par and pbegin

These constructs can be easily implemented by modifying the SHE-based implementation of **pcall.** Their implementations are slightly simpler than that of **pcall,** since there is no need of implementing function application.

(2)
plet and pletrec

Note that local bindings in these constructs are performed sequentially, because the cost of local binding is much smaller than the cost of a stealing action, which is the cost of process creation in the ETC strategy. See below for the cost in the PaiLisp/MT interpreter.

(3)
par-and, par-or and pif

The argument expressions in these concurrency constructs are evaluated in parallel. In the case of the **par-and** expression, when the value of one of the argument expressions becomes *false,* the value of the entire expression will become *false,* and the rest of computation at the point is killed by a kill signal sent from the RM which evaluated the corresponding argument expression. An actual *killing* action is realized by sending/

receiving a **kill** signal. The similar arguments hold for the **par-or** expression, replacing *false* into *true.* The costs of sending and receiving a kill signal are 11 [μsec] and 16 [μsec], respectively, in the PaiLisp/MT interpreter, **pif** is a parallelization of if that the argument expressions e_1, e_2, e_3 are evaluated in parallel; they are evaluated as if (**pbegin** e_1 e_2 e_3) is evaluated in the SHE strategy with the following modification. e_2 and e_3 are stored into the SST_i of RM_i together with their shared object SO, and then the RM_i starts evaluating the expression e_1. If the value of e_1 is *false* then the following actions are taken. If the values of e_2 and e_3 are obtained then the value of e_3 is returned else if e_2 is stolen and evaluated by an idle processor then a **kill** signal is sent to the processor of evaluating e_2. After evaluating e_3 its resultant value is returned as the value of the **pif** expression. If the value of e_1 is not *false* then the similar actions are taken for the expressions e_2 and e_3.

(4)
On other PaiLisp concurrency constructs

Other PaiLisp concurrency constructs are also realized, **pcond** and **pcond#** are implemented in a similar manner with **pif** and **par-and.** Following the SHE-based implementation strategy of **future** given in the reference 7), the SHE-based **future** is realized in the PaiLisp/MT interpreter, **call/pcc** and **call/pep** are the extensions of the sequential **call/cc** and **call/ep** to support *P*-continuation, and they are implemented in the manner explained in the reference 9) and 10).

4.3
Experimental Results of the PaiLisp/MT Interpreter

The PaiLisp/MT interpreter is a parallel Scheme system equipped with multiple evaluation strategies (ETC, SHE, LTC, Sequential Evaluation), implemented on a DEC7000 system with six Alpha Processors under OSF/1 OS. In this section we give some basic costs in the evaluation strategies implemented in the PaiLisp/ MT interpreter, and then we give some results of running several programs on the PaiLisp/MT interpreter.

4.3.1
Basic costs in the PaiLisp/MT interpreter

- The cost of a Scheme primitive (a constant, a variable, and a quote expression) is smaller than 2 [μsec].
- The cost of applying a primitive Scheme function (like **cons, car, cdr, +, −, =, null?, eq?,** etc.) is about 5 [μsec].
- The cost of binding a value to a variable is about 2 [μsec], and the cost of forming a lambda closure is about 2.2 [μsec].
- The cost of function application is about 5 [μsec].
- The cost of creating a shared object is 9 [μsec].
- The cost of stealing a pair of an expression and its shared object from SST is 70 [μsec], while the cost of process creation in the ETC strategy is 60 [μsec].
- The cost of exclusive access to SST by the home processor is 11 [μsec], and the costs that a helper accesses to the Resume Stack and to the SST of the home processor are about 4 [μsec].
- The cost of sending a **kill** signal is about 11 [μsec], and the cost of receiving a **kill** signal is about 16 [μsec].

- The cost of creating a future-value is 15 [μsec], and the cost of the **force** operation to obtain an actual value for a future-value is about 14 [μsec].

4.3.2
Running several programs on the PaiLisp/MT interpreter

The following programs[1] were used for an experimental evaluation of the PaiLisp/MT interpreter.

Fibonacci	(fib n), (pfib n), (ffibl n), (ffib2 n), (ffib3 n)
Tarai	(tarai x y z), (ptarai x y z), (ftarai100 x y z), (ftarai010 x y z), (ftarai001 x y z), (ftarai110 x y z), (ftarai101 x y z), (ftarai011 x y z), (ftarai111 x y z)
Merge sort	(msort d), (pmsort d), (fmsort d)
N Queen	(queen 8), (pqueen 8), (fqueen 8)

Figure 4 shows the results of running these programs on the PaiLisp/MT interpreter, using six processors in the SEQ, ETC, SHE and LTC strategies, where SEQ means *Sequential Evaluation.* In the above benchmark programs **fib**, **tarai**, **msort** and **queen** are sequential programs of evaluating the Fibonacci function, the tarai function, the merge sorting of data and the N Queen problem, respectively, **pfib**, **ptarai**, **pmsort** and **pqueen** are their parallelized versions, using the **pcall** construct, while ffib*i*, ftarai*ijk*, fmsort and fqueen are their parallelized versions, using the **future** construct. Note that the suffices in ffib*i* and ftarai*ijk* indicate the places where **future** is inserted.

From the results of Figure 4 we can observe the following:

- The SHE strategy always runs faster than the ETC strategy.
- The LTC strategy sometimes runs slower than the ETC strategy, and the effectiveness of the LTC strategy depends upon the places where **future** is inserted for parallelization.
- The SHE strategy sometimes runs slower than the LTC strategy, although their differences are small. But the SHE strategy for a program runs faster and more stable than the LTC strategy for the corresponding program.

Thus, we would be able to say that the SHE strategy is a more stable and sound parallel evaluation strategy than the ETC and LTC strategies.

5
PaiLisp/MT Compiler

The PaiLisp/MT compiler is a compiler of translating a PaiLisp program into a RM program, and the resultant RM program is translated into the C language program. Then the resultant C program is executed under the PaiLisp/MT interpreter. The input/output routines, the heap area and the garbage collector of the PaiLisp/MT interpreter is commonly used in the compiler. The PaiLisp/MT compiler consists of two parts:

[1] These programs are available by anonymous ftp through **ftp://ftp.ito.ecei.tohoku.ac.jp/pub/pailisp/benchmarks.tar.gz**

(1) Compiler for Scheme expressions.
(2) Compiler for PaiLisp concurrency constructs.

Compiled codes are dynamically linked under the interpreter. Separate compiling of procedures is possible, even if they mutually invoke and reference each other, and also compiled procedures and interpreted procedures can invoke and reference each other. Note that for PaiLisp concurrency constructs two code generation rules based on the ETC and SHE strategies are available.

On implementing call/cc and call/ep

Besides the above considerations we have to consider how to implement *continuations.* In case of Scheme it is enough to support *sequential continuation,* captured by **call/cc.** However, in implementations of PaiLisp we have to consider two aspects in supporting *continuations.* One is in introducing **call/pcc** to support *P*-continuation, which is an extension of Scheme continuation (*S*-continuation) into concurrency. Another is a restricted version of **call/cc**, written as **call/ep,** to support only a single-use continuation, and similarly a restricted version of **call/pcc**, written as **call/pep**, to support a single-use *P*-continuation[8, 10]. There are a number of strategies of implementing *continuations*[2]. The stack strategy is a commnonly-used technique of implementing continuations, and it is efficient for a program which does not contain any **call/cc** construct. However, **call/cc** implemented in the stack strategy is slow and heavy, since it requires the actions of copying contents of the control stack in capturing and throwing continuations, **call/ep** and **call/pep** are introduced, since **call/cc** and **call/pec** are usually used in a single-use style of continuations in practical programs. They can be efficiently implemented only in handling tag-objects stored in the control stack without copying contents of the stack[10, 11]). Note that the stack/heap strategy[2]) is efficient in implementing **call/cc** and **call/pcc,** since it does not require copying contents of the stack in throwing continuations. But it is not applicable in implementing **call/ep** and **call/pep,** because the tag information will be sometimes lost under the stack/heap strategy.

5.1
Compiler for Scheme Expressions

A Scheme compiler is implemented, slightly extending the Abelson-Sussman's compiler[1]. Let *C* [*E,* ctenv, cont] be the compiler of a Scheme expression *E* to generate the corresponding RM code program under ctenv and cont, where ctenv means the *compile-time-environment,* and cont is a keyword to be used during compilation, cont will be one of return, next and LABEL, as explained below. The compilation rules for major Scheme constructs can be given as follows.

(1)
Compilation rules for quote and a constant

C [(quote *number*), ctenv, any]==>*C* [*number,* ctenv, any] *C* [(quote *boolean*), ctenv, any]==> *C* [*booleant,* ctenv, any] *C* [(quote *TextOfQuotation*), ctenv, return] ==>(assign val '*TextOf Quotation*) (restore cont) (goto cont) *C* [(quote *TextOf Quotation*), ctenv, next] ==> (assign val '*TextOf Quotation*) *C* [(quote *TextOf Quotation*), ctenv, LABEL] ==>(assign val '*TextOf Quotation*) (goto LABEL)

program	SEQ [sec]	ETC [sec]	SHE [sec]	LTC [sec]
(fib 20)	0.655 (0)	–	–	–
(pfib 20)	–	0.767 (21890)	0.167 (50)	–
(ffibl 20)	–	0.519 (10945)	0.187 (51)	0.162 (66)
(ffib2 20)	–	2.011 (10945)	1.121 (531)	1.867 (10945)
(ffib3 20)	–	0.922 (21890)	0.251 (92)	0.210 (173)
(tarai 840)	0.424 (0)	–	–	–
(ptarai 840)	–	0.321 (9453)	0.097 (140)	–
(ftarai100 840)	–	0.231 (3151)	0.152 (137)	0.143 (84)
(ftarai010 840)	–	0.822 (3151)	0.829 (3151)	0.860 (3151)
(ftarai001 840)	–	0.003 (73)	0.003 (22)	0.034 (32)
(ftarai110 840)	–	0.353 (6302)	0.197 (190)	0.191 (156)
(ftarai101 840)	–	0.078 (246)	0.004 (7)	0.094 (74)
(ftarai011 840)	–	0.352 (4323)	0.004 (22)	0.063 (59)
(ftarai111 840)	–	0.151 (2910)	0.004 (30)	0.312 (60)
(queen 8)	2.144 (0)	–	–	–
(pqueen 8)	–	0.613 (11016)	0.389 (194)	–
(fqueen 8)	–	0.522 (5508)	0.411 (75)	0.394 (191)
(msort d)	0.535 (0)	–	–	–
(pmsort d)	–	0.182 (1000)	0.148 (17)	–
(fmsort d)	–	0.166 (500)	0.149 (18)	0.158 (22)

Figure 4: Results of running several benchmark programs on the PaiLisp/MT interpreter

Remark: return, next, LABEL, and any