Parallel Functional Programming

An Annotated Bibliography

*(2nd Edition)**

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
email: `schreine@risc.uni-linz.ac.at`

December 6, 1993

Abstract

This bibliography cites and comments almost 400 publications on the parallel functional programming research of the last 15 years. It focuses on the software aspect of this area i.e. on languages, compile-time analysis techniques (in particular for strictness and weight analysis), code generation, and runtime systems. Excluded from this bibliography are publications on special architectures and on garbage collection unless they contain aspects interesting for above areas. Most bibliographic items are listed inclusive their full abstracts.

*Supported by the Austrian Science Foundation (FWF) contract S5302-PHY "Parallel Symbolic Computation".

Contents

1 Introduction to the 2nd Edition

In October 1993 I announced the availability of the first version of this bibliography on the Usenet news groups `comp.lang.functional` and `comp.parallel`[1]. After that I have received a fairly large number of (mostly encouraging) comments, corrections and supplements.

Hence I decided to use these replies as an opportunity for providing an updated edition of the bibliography that contains citations that were previously missing (and also new citations on recently published papers).

The differences to the previous edition are not large but significant; the bibliography now also includes

- The nested data-parallel language NESL (thanks to Guy Blelloch, Carnegie Mellon University),
- The parallel equational programming language EPL (thanks to Bolek Szymanski, Rensselaer Polytechnic Institute),
- New citations and corrections on SISAL (thanks to Rod Oldehoeft, Colorado State University),
- New citations on the skeleton approach (thanks to Fethi A. Rabhi, University of Hull, and Tore Bratvolt, Heriot-Watt University).
- Citations on work done at the GMD in Bonn on parallel functional programming (thanks to Werner Kluge, GMD).
- The real-time parallel functional language Erlang.

This new edition is available by anonymous ftp from the server `ftp.risc.uni-linz.ac.at` (193.170.36.100) in `pub/reports/parlab` as `fpcbib2.ps.Z`. By several requests, also the BibTeX sources can are available as `fpcbib2.tar.Z`.

Kevin Hammond was so friendly to duplicate (the first and maybe also the second edition of) the bibliography and the associated BibTeX files on the Glasgow University ftp server `ftp.dcs.glasgow.ac.uk` (130.209.240.50) in directory `pub/glasgow-fp/bibliographies`.

As for the previous release, I would be glad to receive any comments, corrections, and supplements.

2 Introduction

For one and a half decade computer scientists have now studied the application of functional languages for the programming of parallel computers. This interest has been motivated by the lack of a global store in functional languages, which allows the independent evaluation of function arguments (horizontal parallelism) as well as the simultaneous execution of a function with the evaluation of its arguments (vertical parallelism).

Parallelizing a functional program is therefore a basically simple task: any subexpression can be evaluated on another processor; no complicated data dependency analysis is necessary to detect parallelism. An open and shut case? Not at all. It turns out that

> *while it is very simple to transform a functional program into a parallel program, it is very difficult to transform it into a program that can be efficiently executed on a parallel computer!*

On parallel architectures, the creation of a task consumes much more resources than the invocation of a function and only pays off if

[1] An earlier attempt in May 1993 had failed unnoticed.

1. the task runtime outweighs creation, communication and synchronization overhead,
2. the system has enough memory to accommodate the new task, and
3. some processor would eventually idle if the task were not spawned.

The first condition gives a lower bound for the granularity of tasks while the second limits the total number of tasks; the third item limits the amount of parallelism that can be actually exploited.

Alternative Architectures versus Stock Hardware

One way to circumvent the first restriction is to develop parallel architectures where task creation is as efficient as function invocation. In this case, both notions coincide and *every* expression can be evaluated as a separate task. This very appealing idea has led to the development of *dataflow machines* and *graph reduction machines* that are optimized for the parallel execution of functional languages.

However, this approach has several practical disadvantages. First, it does not remove other two restrictions that principally apply to all sorts of parallel architectures. Second, most alternative computation models actually blur the distinction between the sequential mode and the parallel mode of computation by making the latter somewhat faster but the former substantially more inefficient.

Today's micro-processors have evolved from the pure von-Neumann model into super-pipelined and super-scalar architectures. They implement the sequential mode of execution in an extremely efficient way; it seems unlikely that alternative architectures will be able to compete with them in the sequential execution mode where, in medium- to coarse-grained parallel programs, most of the computation time is spent.

On the other hand, for fine-grained parallel programs, where alternative computational models might perform better, the actual bottleneck is less processor architecture but more network and communication technology. It is already a substantial problem to transmit data at the required bandwidth from its local memory into the processor, not talking of remote memory access!

In the past implementations of high-level languages on conventional hardware have always turned out to be superior in the long-term[2]. First, they automatically profit from improvements in main-stream technology. Second, it turns out that the benefits of special processors can be also achieved to a good deal by more sophisticated compilation techniques. What was yesterday's special processor, has become today's runtime system and will be tomorrow's compilation model!

Automatic Parallelization versus Parallel Programming

As much research has been devoted to the parallelization of functional languages, it has become more and more clear that

> *the problem of selecting the useful parallelism in functional programs is (almost) as difficult as detecting the possible parallelism in imperative programs.*

This realization has also changed the attitude of researchers concerning the involvement of the human programmer in the parallelization process.

The original idea was doubtless to hide parallelism totally from the programmer. He was supposed to develop a functional program on a sequential computer without caring for

[2] See the fate of Lisp and Prolog machines.

parallelism, to feed the program into the compiler, and to receive an explicitly parallel program computing the same result in shorter time.

This view always suffered from the problem that the solution of a problem can be expressed in many forms which may have very different parallelization characteristics. A simple example is the factorial function whose usual single-recursive formulation is inherently sequential but whose divide-and-conquer form is easy to parallelize.

It is very unlikely that a fully automatic tool will be clever enough to perform such transformations on programs of realistic complexity. A more promising approach therefore assumes that the programmer has already designed a parallel algorithm and formulated it in a functional notion. The task of the compiler is then to translate this notation into an explicitly parallel form.

The programmer himself is therefore responsible for the development of a parallel algorithm. However, he may express his ideas in a high-level notation that is easy to understand, to manipulate, and to reason about, and that prohibits common parallel programming errors (like non-determinism and deadlocks). The compiler may exploit the available parallelism but also generate sequential code.

An even more pragmatic approach has become known as *para-functional programming*. Functional constructs or annotations are introduced that point out those parts of a program where parallel execution may be adequate. In contrast to parallel constructs of imperative languages, these annotations do not carry semantics i.e. they can be added or discarded without influencing the program result[3].

The task of a parallelizing compiler is therefore to free the programmer from low-level details of parallel programming (task creation, communication, and synchronization) such that he may concentrate on the high-level aspects, i.e. the development of the parallel algorithm.

Contents

This annotated bibliography compiles a good deal of the relevant literature on parallel functional programming from the late seventies until today. It focuses on the following topics:

1. Parallel functional programming techniques,
2. Streams, process networks, and functional operating systems,
3. Distributed functional data-structures,
4. Parallel annotations and language extensions,
5. Non-deterministic language extensions,
6. Program analysis techniques (strictness and weight analysis),
7. Compilation and code generation,
8. Abstract machines and runtime systems,
9. Run-time control of parallel functional programs,
10. Implemented parallel functional systems.

This bibliography does deliberately *not* include the following items each of which would require another bibliography of the same extent:

[3]At this point functional languages meet the efforts for the automatic parallelization of imperative languages. The new "High Performance Fortran" (HPF) standard introduces constructs and annotations that allow the compiler to assume that parts of a program are executed in a functional way (i.e. without side-effects).

1. Alternative architectures,
2. Garbage collection.

However, papers in these categories are cited if they contain aspects that fall into one of the areas above.

Each section of the bibliography first explains the problem and puts the listed publications into the general context. The following list of citations is in general ordered according to publication date, but closely related papers are usually cited together.

All comments and annotations merely represent the interpretations and opinions of the author of this bibliography. However, most bibliographic items are listed with their authentic abstracts or, if the paper contains none, short summaries derived from the introductions.

The author would be glad to receive any comments, corrections, and supplements to this bibliography. I am greatly indebted to the work of the Parallel Computation Group at RISC-Linz, in particular to Hans-Wolfgang Loidl, for gathering several of the most recent publications in this area.

3 General Work

Problem Description

In this section, we survey those publications that

- provide an overview on the area of parallel functional programming,
- describe techniques for writing parallel functional programs. or
- present case studies of parallel functional programs.

Since this is an active research area that changes very rapidly, these survey papers can be only evaluated by also considering their publication dates.

Relevant Literature

[Richards Jr., 1984] is a bibliography on functional programming that contains most of the early approaches of parallel implementations of functional languages. [Joy, 1990] is a more recent bibliography that contains some hundred citations on the parallel evaluation of functional programs.

[Berkling, 1975] is one of the earliest papers discussing the application of functional languages for parallel processing. About the same time [Friedman and Wise, 1976, Friedman and Wise, 1978b, Friedman and Wise, 1979] consider parallel argument evaluation and the non-strict construction of data structures are considered as the essential sources of parallelism.

[Kennaway and Sleep, 1982b] surveys several approaches to parallel implementations of functional languages.

[Peyton Jones, 1987] is the standard textbook on the implementation of lazy functional languages by programmed graph reduction and also sketches some parallelization issues. In more detail, [Peyton Jones, 1989] describes the main approaches to parallel implementations of functional languages based on the concept of parallel graph reduction.

[Field and Harrison, 1988] is the standard textbook on functional programming in general and also discusses parallel implementations of functional languages by graph reduction and dataflow. Furthermore, the modelling of process networks by stream-based functional programs is described.

[Kelly, 1989] contains a comprehensive survey on the state of the art in the parallel implementation of functional languages. Typical functional program patterns are given and their parallelization is discussed.

[Martins, 1992] provides an overview on the main proposals for the parallel implementation of functional languages. Several parallel Lisp, dataflow and graph reduction systems are described.

[Rabhi, 1993b] presents some of the techniques used to execute functional programs on multiprocessors. The paper concentrates on parallel graph reduction and on the "algorithmic skeletons" approach and lists some 100 references to the relevant literature.

[Wray and Fairbairn, 1989] discusses the design of an interactive functional program based on streams. The abstract three-instruction-machine TIM is described for the implementation of lazy languages.

[Roe, 1989] discusses several issues of parallel functional programming, such as how far parallelism should be explicitly expressed in the program and how the task size can be controlled.

[Turner, 1981] describes a functional program for the so called "paraffins problem" that relies on streams and the non-strict evaluation of functional languages. [Arvind *et al.*, 1988b] refines the solution using the dataflow language and generates by simulations the parallelism profile of the program.

[Hudak and Anderson, 1988] presents functional solutions of several language session problems presented at the 1988 Salishan Conference on High-Speed Computing.

[Arvind and Ekanadham, 1987, Ekanadham and Arvind, 1987] present another case study on parallel functional programming in Id: a dataflow version of *Simple*, a benchmark program for computational fluid dynamics and heat flow, is developed and compared with the corresponding Fortran solution.

[Arvind *et al.*, 1986, Arvind *et al.*, 1988a] present the Id solutions to several smaller problems and analyze their inherent parallelism by simulation.

[Lee *et al.*, 1988] presents the performance of a set of algorithms written in Sisal and run on multiprocessor Sequent, DEC, and Cray computers.

[Akerholt *et al.*, 1991] describes the implementation of a parallel database manager on the graph reduction architecture GRIP and presents detailed experimental results.

4 Streams, Process Networks, and Operating Systems

Problem Description

Parallel programs are often formulated as networks of parallel processes that communicate by sequences of messages. The parallel execution model of functional languages that is based on the parallel evaluation of function arguments (perhaps concurrently with the function body) seems to fundamentally contradict this view. However, both views coincide if *streams (non-strict lists)* are taken into account.

A functional program like

$$l_1 = f(a,b)$$
$$l_2 = g(a,b)$$
$$l_3 = h(l_1,l_2)$$

where the l_i are non-strict lists may be interpreted as the description of a network of three parallel processes communicating by streams of messages. The process executing h may e.g. consume one message from l_1 and l_2, respectively, and produce one message in l_3.

In non-strict functional languages, it is also possible to apply *circular definitions* to construct process networks with cyclic communication between processes like in

$$l_2 = f(l_1)$$
$$l_1 = g(l_2)$$

Circular definitions allow to express general process networks and manager/worker schemes of computation. Furthermore, they may be applied to construct multiply linked data structures (such as doubly linked lists).

However, by circular definitions *deadlocks* may be introduced, if a stream element depends on other stream elements that have not yet been computed. A particular problem therefore is to guarantee the *productivity* of circular definitions i.e. to guarantee that circular definitions make sense and lead to terminating programs.

The process-oriented view of functional programs has been used for the referentially transparent representation of *input/output* in functional languages (where a functional program communicates with the outside world by request/result streams), for the implementation of *interactive programs* and in particular for the design of *functional operating systems*.

A related problem is how to express erroneous computations in functional programs interpreted as descriptions of process networks. One approach is the introduction of dedicated *error values* that may be propagated through the network.

Relevant Literature

[Henderson, 1980] describes a number of elegant process network solutions to various problems using streams for the communication between processes. Likewise, the textbooks [Field and Harrison, 1988] and [Bird and Wadler, 1988] use non-strict lists to formulate several process-oriented programs.

[Wand, 1980] shows how continuations can be used to model the notion of state in applicative languages and applies the technique to the design of an operating system kernel.

[Karlson, 1980] describes a functional operating system *Nebula* which is based on streams but uses higher order functions to simplify the operating system interface.

[Jones, 1983, Jones, 1984] also describe an stream-based approach to operating systems written in a functional style. The system has been implemented on a physical network of processors. Similar approaches in this direction are [Henderson, 1982], [Abramski and Sykes, 1985] and [Stoye, 1984, Stoye, 1986].

[Haynes *et al.*, 1984] implements a variety of coroutine mechanisms using continuations and higher-order functions. In particular, the design of a stream interface is discussed. Likewise, [Allison, 1987] shows how continuations can be used to simulate coroutines.

[Haynes and Friedman, 1984] introduces the notion of *engines*. An engine is an object that encapsulates a computation and takes a number. If the computation is not finished in the specified number of evaluation steps, the engine returns an engine that, when run, continues the computation. A process scheduler is implemented using engines.

[Jagannathan, 1986] describes an operational model for an abstract dataflow interpreter functional which contains stream-based support for data backup and recovery.

[Turner, 1987, Cupitt, 1988, Cupitt, 1989a, Cupitt, 1989b, Turner, 1990] refine the stream-based approach in the operating system KAOS (Kent Applicative Operating System).

[Hudak and Sundaresh, 1988] suggests a functional I/O model based on *systems*: a program maps a single input system (the state of the environment when the program is called) into a list of output systems (a sequence of state changes). The model is proven equivalent to the stream model and to the continuation model of I/O and it is shown how the HASKELL language implements this approach. Parallel programming paradigms like the Actor model, UNITY, Linda and CSP can be also expressed in this approach.

[Perry, 1988a] discusses several proposals for I/O in functional languages by environment passing, by streams, and by continuations. A new approach based on *result continuations* is proposed: the return value of a program is an operation request (to be carried out by the operating system) and a result continuation that is to be called with the result of the request as argument. The implementation of this idea in Hope+C is presented.

[Perry, 1988b] discusses an operating system design based on result continuations. [Perry, 1992] shows how result continuations can be used to implement coroutines and lays the groundwork for the development of a concurrent functional and object/process-oriented programming language.

[Jones and Sinclair, 1989] discusses the design of an operating system based on a functional language with networks of processes communicating by streams.

[Arya, 1989] describes a functional approach to an animation system displaying sequences of pictures. The system is based on a stream-connected network of processes.

[Thompson, 1990] provides a formal semantics of stream-based interactive functional programs in terms of trace sets.

[Hieb and Dybvig, 1990] generalizes continuations to *process continuations* that can be used in tree-structured concurrent computations to denote the rest of a process from a given point in the process. This can be used to achieve non-local exits without interfering with concurrent computations and to save and restore selected process subtrees.

Other approaches to use continuations in parallel context are described by [Giorgi and Le Métayer, 1990] and [Morrisett, 1991].

[Broy and Dendorfer, 1992] introduces the formalism of higher-order stream processing functions where stream elements may be also functions. Time-stamps are introduced to model higher-order behavior.

[Arvind and Brock, 1984] applies the basic stream model to implement *managers* that control resources shared by several processes.

[Harrison, 1987] introduces the real-time functional programming language RUTH where processes communicated by streams of time-stamped values. Applying the approach proposed by [Burton, 1987], a tree of integers is passed as argument to the program whose leaves evaluate to the current time. Taking real-time information as input data, referential transparency is preserved. [Williams *et al.*, 1990] describes another approach to real-time functional programming.

[Wetherell, 1982] describes how in the dataflow language VAL special error values are used to represent faulty operations.

[Hammond, 1989a] presents a version of the error value method which allows lazy computation with deterministic semantics for divide-and-conquer parallel evaluation even in the presence of errors.

[Morris and Schwarz, 1980] is the first paper discussing the virtues of cyclic list structures (in the context of Lisp programs). It describes describe how recursive object definitions can be used to construct such lists in a purely functional form avoiding explicit pointer updates.

[Kieburtz, 1986] illustrates why recursive object definitions lead to efficient graph reduction programs provided that cyclic data structures are used to represent the recursively defined expressions.

[Bird, 1984] gives several examples for programs over circular data structures that are more efficient than their non-circular counterparts. A technique is described for transforming functional programs that repeatedly traverse a data structure into circular programs that do not. [Takeichi, 1987] shows how curried functions and fully lazy evaluation can be applied for the same purpose.

[Allison, 1989b] describes how circular programs can be used to construct data structures like doubly-linked lists or trees where all leaves are directly linked. [Allison, 1987, Allison, 1989a] gives several examples of programs with circular list definitions.

[Sijtsma, 1988, Sijtsma, 1989] formalizes the notion of *productivity* of recursive list definitions. The author constructs a mathematical model and gives conditions that are sufficient to guarantee that each element of a recursively constructed list can be computed in a finite amount of time. A calculus is developed formalizing the notion of *segment productivity* which is more restricted but allows a large class of functional programs with recursive list definitions to be proven deadlock-free in a (semi-)automatic way.

[Schepers, 1991] describes the *GRAPH* system, a tool for the specification, analysis and execution of functional processes where communication is realized via streams.

[Loidl, 1992c] gives a comprehensive survey on the literature in the area of circular programs and circular data structures.

5 Distributed Data Structures

Problem Description

Linear lists are the dominant data structure in functional programming. However, by their sequential structure they may be less suitable for expressing parallel algorithms. Other forms of data structures have been proposed that better meet the demands of parallel computation by exposing the parallelism in functional programs. A special form of of data structures related to parallelism are *multisets/bags* that are especially interesting in relationship with non-deterministic computations (see Section 7).

Relevant Literature

[Keller, 1980] proposes *concatenations* as an alternative representation of sequences of data: each cell of the data structure contains two links each of which references either an atom, or a tuple of elements or a concatenation that is constructed in the same way. The proposed representation is especially suitable for divide and conquer algorithms.

[Harrison III, 1986] proposes a representation of lists as linked arrays where each array element contains either a list element or a pointer to another array denoting the following list elements.

[Li and Hudak, 1986] proposes a list compaction method that is suitable in a parallel context when several list are constructed simultaneously from the same heap. A data representation is used where sequences of data are stored as linked vectors i.e. each cell is of arbitrary size and may contain besides a reference to the next cell an arbitrary number of elements.

[Arvind *et al.*, 1986] introduce I-structures as data structures for the communication and synchronization between parallel processes in the dataflow language Id. An I-structure is a single-assignment array where each slot is extended by a presence bit that indicates whether a value has already been written into this slot. Processes that try to read an undefined element are implicitly delayed. I-structures are therefore able to represent high-level non-strict data structures such as streams and can their application yields elegant programs with much inherent parallelism (e.g. *wavefront* programs).

[Wise, 1987] suggests 2^d-ary trees to represent d-dimensional arrays; examples are matrix operations ($d = 2$) and presents several algorithms from matrix algebra (in particular a pivot step algorithm) that operate with this representation.

[Kuchen, 1989] proposes a new data structure *distributed applicative arrays* whose elements are distributed among processors. If a new array is created, the current processor allocates some part in its local memory and tells its neighbor processors which parts of the array they are in charge of. These processors may again forward parts of the allocation requests to other processors. Access requests are thus either handled locally or forwarded to the other processors.

[Hurson and Lee, 1989] proposes a new representation of data structures in data flow environments. *Hybrid structures* are represented by tables where one field references the original array while another field references a table of modifications on the array. If by an update operation *upd(a,i,v)* a new array b is constructed which is a copy of a but contains the value v at index i, the original array can be shared (preserving referential transparency) and only a new modification table has to be constructed.

[Maaßen, 1992] reviews several proposals of applicative data structures for parallel computing. Three new data structures *sequence*, *table* and *set* are introduced and possible implementations are discussed.

6 Para-Functional Programming

Problem Description

Since the automatic parallelization of functional languages is rather difficult, many language extensions have been suggested in order to give the programmer the possibility to be more explicit about his parallelization ideas and wishes, respectively. According to their influence on the functional subset of the language, we may classify these approaches as follows:

1. **Explicit Parallel Constructs:** In functional languages with imperative extensions (like Lisp or ML) the well-known parallel extensions of sequential languages (e.g. message passing) may be introduced. Parallel programming thus takes place on an imperative level but may profit from the functional language subset.

2. **Functional Parallel Constructs:** Even in not purely functional languages, it may be the case that all parallel constructs assume that the involved functions do not cause any side-effects. Provided the correct application of these constructs (that the programmer is in charge of), this yields a rather functional programming style.

3. **Data Parallel Constructs:** On SIMD (Single Instruction Multiple Data) architectures, these parallel language extensions usually have to be restricted to certain forms of "data parallelism" where the same operation is applied to all elements of an aggregate (e.g. array).

4. **Parallel Annotations:** If the basic language is purely functional, parallel language extensions turn into mere program annotations that do not influence the correctness of a result but only the efficiency of its computation. The programmer may insert or remove such annotations without any risk.

The last category is often denoted by the term "Para-functional Programming" [Hudak, 1986c]; however, all categories but the first one actually lead to a common programming style with similar characteristics.

"Algorithmic skeletons" are a related term coined by [Cole, 1989]. It denotes a set of predefined higher order functions that represent the structure of typical parallel algorithms. These algorithmic skeletons (that can be parameterized by the concrete application functions) may be efficiently parallelized and mapped onto the actual architecture.

A related category are the **single-assignment languages** that have evolved from imperative languages by restricting the assignment operation and thus yield functional characteristics. Modern languages of this type are actually equivalent to functional languages; the only (but mainly syntactic) difference is their explicit support of iterated loops.

Explicit Parallel Constructs

[Prini, 1980] describes the parallel Lisp dialect *PARLAM* where the arguments of a function are automatically evaluated in parallel with the execution of the function body. Explicit synchronization constructs are applied to wait for process results. A non-deterministic construct allows to apply speculative parallelism.

[McKay and Shapiro, 1980] describes a package of Lisp functions called *MULTI* with low-level primitives for multi-processing.

[Model, 1980] proposes multiprocessing via intercommunicating Lisp systems.

[Cohen *et al.*, 1983] describes a Lisp dialect *PLISP* with annotations for the parallel evaluation of function arguments concurrently with the body of a function.

[Brandenburg *et al.*, 1988] describes a parallel Lisp system for Intel Hypercubes with *remote evaluation* constructs for process creation and *streams* (in the sense of message channels) for communication.

[Dimitrovsky, 1988] describes the parallel dialect *ZLISP* for the NYU Ultracomputer prototype. The language has primitives for explicit process creation and special parallel data structures (e.g. queues) for communication.

[Weber, 1989] describes the package *dLisp* for the distributed evaluation of Lisp programs on local area networks. The package provides facilities for (speculative) parallelism and supports manager/worker parallelism with communication by message channels.

[Zorn *et al.*, 1987, Zorn *et al.*, 1988, Zorn *et al.*, 1989] describe the concurrent Lisp dialect *Spur Lisp* with explicit process creation constructs. Communication and synchronization is based on *mailboxes*. On a higher level, the language provides futures in the style of Multilisp.

[Takeuchi, 1988] describes a concurrent Lisp dialect *TAO* for the ELIS Lisp machine with semaphores and mailboxes for synchronization and communication.

[Iwasaki, 1988] describes the concurrent Lisp dialect *mUtilisp* with explicit process creation and communication by message passing.

[Hammer and Henties, 1990, Ilmberger Hermann, 1991] describe *EDS Lisp*, a parallel Lisp version for the European Declarative System. The language is based on Multilisp futures and on explicit communication by mailboxes.

[Koorey Willcock, 1992] describes a concurrent Lisp dialect *Tlisp* for transputer systems with constructs for explicit process creation and synchronization.

[Reppy, 1989, Reppy, 1991b, Reppy, 1991a] describe the concurrent ML dialect *CML*. The language is based on the explicit creation of threads and communication by message channels. First-class synchronous operations are provided as a mechanism to build higher-level synchronization and communication abstractions (such as Multilisp futures). The implementation is based on coroutines and therefore only quasi-parallel.

[Cooper and Morrisett, 1990, Morrisett and Tolmach, 1992] describe another approach of adding threads to ML. There are primitives for explicit process creation and synchronization by mutual exclusion locks and signals. On top of these operations, higher order communication and synchronization primitives are built. The system was implemented on shared memory multiprocessors.

[Matthews, 1991] describes a distributed implementation of Standard ML. [Berrry *et al.*, 1992] gives a formal semantics of ML concurrency primitives.

[Jorrand, 1984, Jorrand *et al.*, 1985] introduce the language FP2 (Functional Parallel Programming) for describing networks of communicating processes. Each process is defined by its current state, a set of ports through which mesages flow in and out and a set of rewrite rules that describe the change of the process state on a communication. If the patterns of several rewrite rules match the current state and the requested communication type, one rule is non-deterministically selected and applied. Processes are created and combined by operators and forms that define an algebra of processes. The resulting topology is static i.e. FP2 does not provide facilities for dynamic process creation.

[Schaefer and Schnoebelen, 1987] describes the design of a pipelined event-driven simulator using FP2 and a proof methodology for FP2 programs. [Kurfeß *et al.*, 1990] describes the design of an abstract parallel inference machine LCM (Logical Connection Machine) based on FP2.

[Glück and Demuth, 1987] gives a similar approach for describing process and communication structures where FP operators are used to combine Occam processes.

[Armstrong and Virding, 1991, Armstrong *et al.*, 1991, Armstrong *et al.*, 1992, Armstrong *et al.*, 1993] describe the parallel functional language *Erlang* that uses explicit process creation and message passing for communication. The language has been designed for real-time distributed telecommunication (telephony) applications.

Data Parallel Constructs

[Steele Jr. and Hillis, 1986] describes *Connection Machine Lisp* a data-parallel SIMD dialect of Lisp. The datatype *xapping* is introduced which is an unordered set of index/value pairs. There are several operations to create, combine, and manipulate xappings. An FP-like "apply-to-all" operator can be used to apply a function to all elements of the xapping in parallel. A "fold" operator combines the elements of a xapping in a non-deterministic order into one.

[Merrall, 1991b, Merrall, 1991a, Padget and Merrall, 1991] base an SIMD implementation of EuLisp on the data structure *paralation* [Sabot, 1988] which is a virtual processor set. Instances of this paralation are fields where each element is stored on a separate processor and where to each element the index of the corresponding processor is associated. Operations on these fields can be performed in parallel by each processor.

[Walinsky and Banerjee, 1990] describes an automatically parallelizing FP compiler that exploits data-parallelism on FP sequences and generates code for SIMD machines.

[Jouret, 1991] extends a functional language by an aggregate type and a set of primitive monolithic operations on this type. Communication and computation are clearly separated (by a higher-order operation `imap` and several shuffle operations) such that transformation rules can be easily applied.

[Hill, 1993] defines a data-parallel extension *DPHaskell* of the lazy functional language Haskell. A parallel data type POD (parallel object with arbitrary dimension) is introduced which represents an infinite set of index/value pairs. Parallel algorithms are expressed by POD comprehensions were elements are drawn from generators lazily (i.e. only when required) to avoid a combinatorial explosion of parallelism.

[Blelloch, 1993, Blelloch *et al.*, 1993] describe *NESL*, a language that in contrast to the other data-parallel approaches also allows nested data-structures and *nested parallelism.* The language also provides mechanisms for calculating the asymptotic runtime of the program on various parallel machine models e.g. the parallel random access machine.

[Hammarlund and Lisper, 1993] discusses the relationship between data-parallel and functional programming. Mathematical definitions of data-parallel contructs are presented by viewing data parallel objects as tabulated functions that are stored in a distributed fashion.

Functional Parallel Constructs

[Halstead, 1984, Halstead, Jr., 1985, Halstead, Jr., 1986, Halstead, 1986, Halstead, 1987] describe the Scheme dialect *Multilisp* that provides the *pcall* constract for parallel argument evaluation and the *future* construct that allows concurrency between the computation of a value and the use of that value. *future* starts a process evaluating an expression but immediately returns a reference to the result which is indistinguishable from the actual result. Any strict operator checks whether its argument is a future or not and blocks when the corresponding process has not yet terminated. Multilisp was implemented by a byte-code interpreter on the Concert multiprocessor built at MIT; this implementation was relatively slow.

[Miller, 1987, Miller, 1988] describes another interpretative implementation of the MultiLisp variant *MultiScheme* for the BBN Butterfly machine. [Swanson *et al.*, 1988] describes a parallel version of Portable Standard Lisp for the Butterfly processor which also relies on the `future` construct.

[Kranz *et al.*, 1989, Kranz *et al.*, 1988] introduce *Mul-T*, a variant of MultiLisp implemented on the Encore Multimax shared memory multiprocessor. The system uses a highly optimizing compiler that generates machine code; its performance is 100 times faster than the other implementations. Futures are automatically inlined if the length of the task queue exceeds a threshold.

[Halstead, 1988] surveys several research directions for the further development of Multilisp including support for *speculative computations* and *lazy futures*.

[Katz and Weise, 1988] shows how the concept of continuations can be cleanly combined with the futures construct of Multilisp.

[Gabriel and McCarthy, 1984, Gabriel and McCarthy, 1988, Goldman and Gabriel, 1988b, Goldman and Gabriel, 1988a, Goldman and Gabriel, 1989, Goldman *et al.*, 1988] describe the parallel Lisp dialect *Qlisp*: The `spawn` construct for process creation returns a future, but in contrast to Multilisp the application of a synchronization construct is required to get the actual value of a future. The `qlet` construct evaluates several variable bindings in parallel. The evaluation of the body expression may optionally proceed concurrently with the evaluation of the bindings, but then the programmer has to explicitly apply the synchronization construct.

The notion of futures is also generalized in two ways: First, the results of several processes may be associated to a single future; as each process finishes, it calls a function that combines its result with the previous value of the future. Second, speculative parallelism is supported by providing an end-test predicate to a combining future. When the value computed by a process satisfies this test, all not yet finished processes associated to the future are killed.

The `qlambda` construct is used to create a closure for a process. This closure may be used by several other processes to transmit *sequences* of values to the process and to receive *sequences* of results. It thus simulates a form of stream-based communication.

`spawn`, `qlet`, and `qlambda` have an extra argument signalling whether new processes shall be actually spawned or evaluations shall be performed locally. This can be used for controlling the amount of parallelism in divide and conquer programs. For the same purpose, there is a predicate saying whether the process queue is empty.

[Pehoushek and Weening, 1988] describes techniques for low-cost process creation and dynamic partitioning of Qlisp programs.

[Ito and Matsu, 1988] describe the language *PaiLisp* that has parallel constructs similar to MultiLisp and to Qlisp. Several non-deterministic constructs are used for speculative parallelism. The translation of all constructs into a kernel language is described.

[Kessler and Swanson, 1988] describes the language *Concurrent Scheme* for distributed memory multiprocessors. Concurrency is created by a `future` like construct. The language introduces *domains* which are monitor-like objects which provide the basic mutual exclusion semantics and an abstraction for different memory hierarchies.

Parallel Annotations

[Burton, 1984] introduces three *application annotations* $@_E$, $@_L$ and $@_P$ for the eager, lazy, and parallel application of a (curried) function to its argument, respectively. These annotations give the programmer complete control over the reduction order of a functional program.

[Burton, 1987] suggests a set of annotations to control the argument passing and evaluation mechanism of functional languages: *value* denotes the eager evaluation of function arguments while *name* denotes their lazy evaluation. *speculation* triggers the non-strict concurrent evaluation of a function argument. *anywhere* lets an expression be evaluated on some arbitrary processor while *at* specifies a processor by the location of data. The *amb* operator non-deterministically returns one of its arguments and does not return only in the case that the evaluation of both arguments does not terminate.

[Hudak, 1986c, Hudak and Smith, 1986] coin the term *para-functional programming* for functional programming with (semantics-preserving) parallelization annotations exemplified by the language ParAlfl. *Mapped expressions* are annotated with the identifier of the processor where evaluation takes place. *Eager expressions* are used to explicitly override the lazy evaluation strategy which is default.

[Hudak and Smith, 1986, Hudak, 1986b] formalize the semantics of para-functional programming languages introducing the notion of *execution trees* to capture the evaluation history of a program.

[Hudak and Anderson, 1987] applies partially order multisets (pomsets) to define the operational semantics of parallel functional programs under different evaluation strategies. Based on this model an annotation language is sketched that allows the programmer to precisely control the evaluation of the program.

[Hudak, 1991] refines this annotation language: each expression can be annotated by a *schedule* defining a partial order of *events*, of which there are tree kinds for every labelled expression `l@e`: `Dl` denotes the demand, `^l` denotes the start, and `l^` denotes the end of `e`'s evaluation, respectively. Two schedules `s1` and `s2` can be composed sequentially as `s1.s2` (denoting that every event in `s1` must precede every event in `s2`) or concurrently as `s1|s2` (without any any additional constraints on the order of events besides those in `s1` and `s2`). *Mappings* specify on which processor an expression shall be evaluated.

[Kewley and Glynn, 1989] describes *strictness annotations* for the parallel implementation of the Hope+ language. The programmer may explicitly specify the degree of evaluation of an argument and also enforce the evaluation of an argument concurrently with the function body.

[Roe, 1989] suggests a binary function *seq* that evaluates its first argument and returns its second argument and a function *par* that spawns a task for the first argument and returns its second.

[Kelly, 1989, Huang *et al.*, 1991] describe an annotation language where the programmer may explicitly specify the partitioning of a functional program into a set of processes. Only static process networks are considered whose parameters are fixed at compile-time.

[Livezey and Muntz, 1989] describes the stream processing language *ASPEN* that applies simple parallelization annotations.

[Maranget, 1991] describes a parallel implementation *GAML* of Lazy ML for shared memory multiprocessors where parallelism is introduced by fork annotations inserted by the programmer.

[Nöcker *et al.*, 1991a] defines the language *Concurrent Clean* that uses simple strictness annotations and parallelization annotations. [Achten, 1991] defines a more expressive annotation language that allows explicit process mapping.

[Glauert, 1992] describes a functional language which is a based on a combination of the λ-calculus by primitives for process spawning and channel-based communication in the style of CCS. The model has not yet been implemented.

Algorithmic Skeletons

[Cole, 1988, Cole, 1989] propose the use of *skeletons* i.e. higher-order functions designed to capture typical parallel computation patterns, e.g. divide-and-conquer. Parallel programs are created by combining instances of these skeletons for which efficient parallel code can be generated.

[Darlington *et al.*, 1991, Darlington *et al.*, 1993a, Darlington *et al.*, 1993b] describe a programming methodology based on skeletons and program-transformation techniques. General functional programs are transformed into compositions of known skeletons and programs are converted from one skeleton to another, adapting it to the characteristics of the underlying architecture.

[Bratvold, 1992] discusses how to automatically identify algorithmic skeletons in functional programs and to extract the useful parallelism. Several experimental results of manually generated parallel code are given. [Bratvold, 1993, Michaelson *et al.*, 1993] present the *SkelML* compiler that exploits parallelism in certain higher order functions and in function composition and generates parallel Occam2 code.

[Schreiner, 1992] discusses the parallelization of typical program patterns arising from programs operating on list structures.

[Rabhi, 1993b, Rabhi, 1993a, Rabhi and Schwarz, 1993] propagate the use of algorithmic skeletons as the "paradigm-oriented" approach to the design of parallel programs. They concentrate on process networks and static iterative transformation algorithms.

Another reference to the skeleton approach is the Ph.D. thesis [Busvine, 1993].

Single Assignment Languages

[Ashcroft and Wadge, 1977, Wadge and Ashcroft, 1985, Ashcroft *et al.*, 1991] describe the single assignment language *Lucid*. The order of statements in Lucid programs is irrelevant, in fact, the language can be seen as a first-order functional language with a monomorphic type system. Special loops provide a syntactic sugar for tail-recursion. The language has been implemented on several multiprocessors.

[Chen, 1985, Chen *et al.*, 1991] describe the dataflow language *Crystal*. Crystal is a purely functional language that provides the special data types *index domains* which embody the geometric shape of data structures and *data fields* which generalize the notion of distributed data structures, unifying the notion of arrays and functions. The language has been implemented on several multiprocessors including hypercube systems.

[Szymanski, 1987, Spier and Szymanski, 1990, Szymanski, 1991a, Szymanki, 1993] describe the "Equational Programming Language" EPL that is targetted for scientific applications. EPL programs are recurrent equations that define the elements of multidimensional arrays (or more general structures). Equations can be annotated with name of virtual processors indexed by the equation subscripts to assist the compiler in program parallelization. Code is generated for several SIMD architectures, shared memory machines and message passing computers.

[McGraw *et al.*, 1985, Oldehoeft and Cann, 1988, Feo *et al.*, 1990, Skedzielewski, 1991, Böhm *et al.*, 1991, Cann, 1992] describe the single assignment language *SISAL* (Streams and Iterations in a Single Assignment Language). SISAL can be seen as a strict first-order functional language with a monomorphic types system. The language provides loops and arrays which are the most important source of parallelism. An explicit stream data type is used for expressing pipelined parallelism. The language is explicitly targeted for scientific computation and has been implemented on data flow computers, shared memory multiprocessors, vector-supercomputers and SIMD machines.

[Nikhil, 1987a, Nikhil, 1987b, Nikhil, 1988, Ekanadham, 1991] describe the dataflow language *Id*. Id is a higher-order polymorphic functional language with a non-strict but non-lazy (i.e. *lenient*) evaluation semantics i.e. (conceptionally) each expression is evaluated by a concurrent process. The language includes *I-structures* i.e. single assignment arrays on which other non-strict data structures (e.g. streams) can be built. Loops can be used for processing these arrays, but the language also provides high-level array comprehension. Id is the standard language of the MIT dataflow project and has been implemented to several dataflow machines.

7 Non-Determinism

Problem Description

One characteristics of functional languages is their simple semantics denoted as *referential transparency*: the result of a function call is uniquely determined by the function arguments (and does not depend on an implicit state like the program store), which allows to apply equational reasoning to any part of the program.

Consequently, functional languages give a much larger degree of freedom than imperative languages concerning the order in which parts of a program have to be evaluated. The constraints derived from the function arguments only express a *partial order* of program operations; any (parallel) evaluation strategy satisfying these constraints yields a deterministic result.

However, there are cases when even the data dependencies derived from a functional program *overspecify* a program result: e.g. if a binary operation f is associative and commutative, then

$$f(w, f(x, f(y, z))) = f(f(w, x), f(y, z)) = f(f(w, y), f(x, z))$$

for any w, x, y, z but a functional program has to be restricted to one particular combination.

This overspecification may have essential consequences on the runtime of a parallel program. Depending on the order in which the arguments w, x, y, z become available and depending on the formulation of the algorithm, the degree of parallelism between different processes may vary very much.

A solution to this problem is to introduce *non-deterministic* operations into a language such that the course of the computation depends on runtime-dependent information like the order in which argument evaluations terminate. However, by non-determinism the referential transparency of a language goes lost and simple equational reasoning is not possible any more.

A possible compromise between both extremes is the introduction of *local non-determinism* into a language such that the result of an operation is unique even with non-deterministic subcomputations. However, then the problem is how to encapsulate non-deterministic computations such that equational reasoning can be applied to the rest of the program.

Anyway, by non-determinism also the question of *fairness* is raised. Essential program properties may depend on the way in which the results of subexpressions are selected, e.g. if the elements of two streams are non-deterministically merged into a single result stream.

Relevant Literature

[McCarthy, 1963] introduces the binary *amb* operator that non-deterministically returns one of its arguments and whose result is only undefined if both arguments are undefined.

[Henderson, 1980] uses the *or* operator (another name for *amb*) and the constant *none*. *or* chooses its return value in a "magical" way such that the evaluation of *none* constants is avoided (if this is possible at all). With this feature backtracking algorithms can be easily implemented.

[Friedman and Wise, 1980] defines the *frons* operator that adds an element to a list at a position determined by the time its value becomes available. [Henderson, 1982] suggests a *merge* operator that non-deterministically interleaves two streams in the order in which their elements become available.

However, all these operators destroy the referential transparency of a functional program and the formal semantics becomes complex [Søndergaard and Sestoft, 1988]. The parallel or operator *por* that returns true when the first argument has evaluated to true and its counterpart *pand* do not suffer from this deficiency but their usefulness is limited.

[Stoye, 1984] suggests an referentially transparent alternative to non-deterministic functions: a program is a function from a (potentially infinite) list of input events to a list of output events; the environment providing the input list performs non-deterministic operations, but the program itself is fully deterministic. [Turner, 1987] refines this approach in the KAOS operating system.

[Hudak, 1986a] gives a new solution to the aggregate update problem involving non-determinism and concurrency. A constract *choose-one* is introduced that returns non-deterministically an element of a list and the list with the chosen element removed. Based on this primitive, the construct *apply-all* applies a list of functions in a non-deterministic order to an argument. It is shown how this construct yields to elegant and efficient programs.

[Burton, 1988] proposes to place non-determinism in *pseudo-data* to allow programs depend on external events. A program is passed an infinite tree of two-valued decisions that may be fixed at runtime thereby permitting non-determinism. Once fixed, a decision remains unchanged, so equivalent expressions must always have the same value.

[Hudak and Sundaresh, 1988] introduces non-determinism through the HASKELL model of input/output where a program receives the state of the environment and returns a sequence of system requests. The system request *amb* returns depending on the environment one of two values.

[Jones and Sinclair, 1989] describes a functional operating system where the *amb* operator is used to express non-deterministic computations.

[Burton, 1989, Burton, 1991] describes an approach to handle non-determinism by a special type *improving* and a speculative *max* function that starts a parallel process for its second argument and returns its first argument as long as the the second argument is not yet available. After the process has terminated, the maximum of both arguments is returned. Using this primitive, algorithms with determinate result but indeterminate behavior can be implemented.

[Hughes and O'Donnell, 1989] introduces a *set data type* whose elements are the possible values of a non-deterministic expression. However, the set is represented only by one of its elements (whose evaluation has terminated first). Since the only operations allowed on this set are the singleton former, union, Cartesian product, and the map operation, equational reasoning about the program is still possible. However, this approach hides the problem more than actually solving it. At last at the termination of the program, the element of the set has to be selected and a non-deterministic choice is required.

A more useful approach is taken by [Clarke, 1990]: The author investigates two classes of associative commutative operations which may be used to exploit the concurrency derived from indeterminate behavior: *simple aggregates* encapsulate the evaluation order indeterminacy implicit in strict associative commutative operations and have no effect on semantics. *Early Readable Aggregates* are aggregates from which information can be read before the evaluation is completed. This reading is restricted in order not to yield non-deterministic program results.

[Marino and Succi, 1989] proposes several data structures that are better suitable for parallel programs than the sequential list structures: *bags* (i.e. unordered collection of objects) with a non-deterministic access function, a folding operator and a mapping operator and *limited functions* with finite enumerable bags as domains. It is shown that programs over bags contain more inherent parallelism than programs over lists.

[Jayaraman and Plaisted, 1987] proposes the language SEL based on equations and sets for functional programming. The user is in charge of establishing a deterministic program semantics by giving annotations whether deterministic or non-deterministic rewriting is to be applied to certain rules.

[Barth *et al.*, 1991] suggests *M-Structures* to express non-deterministic behavior in a non-strict functional language. Each element of an M-structure is tagged with a state bit indicating whether the element is empty or full. Write operations set this bit, read operations unset it, read operations on an empty element are delayed. Using this implicit form of synchronization atomic modification operations can be defined and non-deterministic computations can be expressed. Barriers are used to localize the effects of non-deterministic computations to a binding block. It is shown that this approach yields in several cases to programs that are more efficient, more parallel and more declarative by not overspecifying the execution order.

[Banatre *et al.*, 1988, Banatre and Le Metayer, 1989, Banatre and Le Metayer, 1990] introduce the Γ (GAMMA) language which is based on the non-deterministic manipulation of multisets (bags). The language uses the model of a *chemical solution* in which floating molecules can interact with each other according to reaction rules. Technically, a program is a set of rules that are applied to the element of the multi-set in a non-deterministic way. Any tuple of elements that fulfills the prerequisites of a rule may be replaced by an element as specified by the rule. The program terminates when no more rule can be applied.

Based on the Γ model, [Berry and Boudol, 1990] defines the semantics of an *chemical abstract machine*. Molecules are presented in a systematic way as terms of algebras and the classification of rules is refined.

[Kuchen and Gladitz, 1992] describes the efficient distributed implementation of bags for several forms of Γ operations. The paper gives several algorithms for the binary fold operation and extend the results to more general schemes.

8 Program Analysis

Problem Description

By the static analysis of a program we denote the process of deriving information on the runtime behavior of a program without actually executing it (in particular in an automatic way as by a compiler). This is of especial importance for the generation of efficient code for functional languages , since these languages operate on a rather high-level of abstraction. In the context of parallel computation, there are two questions of particular interest:

1. **Strictness Analysis** addressing the question when it is *safe* to evaluate an expression in parallel[4].
2. **Weight Analysis** addressing the question when it is *efficient* to evaluate an expression in parallel.

[Lester, 1989] describes the technique of *stacklessness analysis* that is able to determine the maximum size of a stackframe in parallel graph reduction. Using this technique, a node which requires evaluation can be created with a stack frame large enough to evaluate all tail recursive calls that may occur in the reduction sequence.

Other analyses are *sharing analysis* and *update analysis* that try to answer the question when a substructure is shared by several other structures i.e. whether a structure can be destructively updated or not.

Strictness Analysis

Problem Description

A large amount of theory has been developed on the compilation of lazy functional languages. In case of sequential machines, the problem is to replace lazy evaluation (which is inefficient) by an eager strategy where values may be immediately reduced to normal form without constructing a graph representation. In case of parallel machines, the problem is to replace lazy evaluation (which is sequential) by a parallel strategy where a task may be started to reduce a value before the value is actually required.

In both cases, the semantics of the program must not be changed i.e. the program must return the same amount of information under a modified evaluation strategy as when reduced by a lazy evaluator. Another demand is usually that the evaluation should not waste any work i.e. it should require not more steps than under the lazy strategy.

An equivalent problem in the eager dataflow model of execution is to increase the granularity of tasks without changing the non-strict program semantics. A dataflow graph which contains many concurrent nodes may be implemented by a sequential thread of execution if all intermediate tokens are necessary to trigger the generation of the result token (however, we are not aware of any effort to apply strictness analysis to the optimization of dataflow programs).

In order to perform this kind of optimizations, the notion of *degree of evaluation* has to be formalized and calculi have to be developed that allow to reason about this notion in an automatic way. In its most simple form, the problem is to decide whether a function f is strict i.e. does not return a result when presented an undefined argument, formally

$$f(\perp) = \perp$$

[4] Only in the case of lazy languages. In the case of eager (dataflow) languages with non-strict semantics, the equivalent question is to decide when it is safe to evaluate an expression *sequentially*.

where $\perp$ represents the undefined value. If above equation holds, any argument a of f may be evaluated (respectively a parallel process evaluating a may be started) before f is called: either the evaluation of a terminates or f cannot return a result anyway.

There are two major methods dealing with the problem of strictness analysis:

1. **Abstract Interpretation:** A non-standard semantics A is introduced that abstracts program values into abstract domains that describe how far an object is evaluated. For each function f an abstract version f_A is generated whose result can be approximated by upper bounds. If this approximation yields

   $$f_A(0) = 0$$

   where 0 is the abstraction of $\perp$, we can deduce that f is strict in its argument. This is a *forward analysis* answering the question how much result information can be derived from a certain amount of argument information.

2. **Projections:** The notion of projections (information-removing functions) comes from domain theory and is applied to formalize the *context* of a function application. If for a function f and two projections α and β

   $$\alpha \circ f = \alpha \circ f \circ \beta$$

   (denoted by $f : \alpha \Rightarrow \beta$) holds, then f may be replaced by $f \circ \beta$ in any context α (i.e. under the assumption α is applied to the result of f). f is translated into a *projection transformers* $\bar{f}$ that return for every context α a safe argument projection $\beta = \bar{f}(\alpha)$. This is therefore an approach to *backwards analysis* answering the question how much argument information is sufficient to derive a certain amount of result information.

The techniques seem complimentary but they have actually different properties (e.g. backwards analysis is not yet able to handle higher-order programs but seems to deliver more accurate information for first-order programs). The relationship between different strictness analysis methods is not yet fully understood.

The problem gets more complicated if *non-flat domains* are considered. While flat objects (e.g. integers) are either evaluated or not, non-flat objects (data structures) may have different degrees of evaluation where e.g. the head of a list is already available while the tail has not yet been computed.

An similar but not identical problem is the treatment *function domains* where functions are first-order values may be passed as arguments of other functions. The strictness analysis of a higher-order function then depends on the strictness analysis of its arguments.

Another complication is the analysis of *polymorphic functions* where the type of the arguments is not uniquely defined and the analysis has to cope with less information than for monomorphic functions.

The compile-time evaluation of abstract functions respectively projection transformers heavily relies *fixpoint iteration* over finite semantic domains. The efficient implementation of this process is therefore crucial for the analysis.

Abstract Interpretation

The pioneer paper [Cousot and Cousot, 1977] introduces the technique of abstract interpretation and its mathematical foundations.

[Mycroft, 1986] explains how abstract interpretation can be applied to the analysis of functional languages. In particular, a formal explanation of strictness analysis is presented for first order functions in flat domains. [Mycroft and Jones, 1985] gives a formal framework of abstract interpretation based on logical relations.

[Clack and Peyton Jones, 1985] gives an informal introduction into strictness analysis by abstract interpretation and presents an efficient representation of abstract functions for fixpoint iteration. The method is presented for first-order programs in flat domains.

[Peyton Jones, 1985] discusses how to apply the results of strictness analysis to generate parallelism. [Hankin *et al.*, 1988] describes how to apply the results of strictness analysis by abstract interpretation to generate parallel combinators from a functional program.

[Maurer, 1985] extends the approach to higher order programs using special λ-expressions (*need expressions*) to represent strictness information. In order to calculate these expressions, certain restrictions have to be made which yields to weak approximation results.

[Burn *et al.*, 1986] provides a mathematical foundation for the strictness analysis of higher-order functions in flat domains. The abstract domain is extended to capture higher-order behavior and the soundness of the approach is proved.

[Bloss and Hudak, 1986] discusses a generalization of strictness analysis to determine what happens before respectively after an expression is evaluated. Four non-standard interpretations are introduced that return the sets of variables whose values will/may be demanded before/after a given expression is evaluated. The analysis distinguishes between the first use of a variable (where evaluation is triggered) and further uses (where the object is already evaluated).

[Wadler, 1987] introduces an abstract domain for the strictness analysis of functional programs over non-flat domains. The domain consists of four points describing various levels of information of a list l: l is totally undefined, l has some undefined tail, l is a finite list with at least one undefined element, or l is a finite list with all elements defined.

[Hudak and Young, 1986] gives a higher-order strictness analyzer for programs in untyped lambda calculus. A set-theoretic characterization of strictness is used that permits treatment of free variables permitting a broader range of interpretations.

[Young, 1989] describes several semantic analyses of an untyped functional language based on abstract interpretation: strictness analysis, termination analysis, operation count analysis, and *collecting interpretations*. The collecting interpretations technique is used e.g. to optimize functions that are called in different contexts (e.g. higher-order functions with different function arguments).

[Burn, 1990c, Burn, 1990b] introduce new domains that cover a notion of *head strictness* which gives extra information about the head of a list. These are a followup papers of [Wadler and Hughes, 1987] which show that not only projections are able to give head strictness information.

[Abramsky and Hankin, 1987] contains several papers on the abstract interpretation of declarative languages.

Fixpoint Iteration

[Clack and Peyton Jones, 1985] introduces *frontiers* (bounds for the argument domain) as the representation of an abstract functions for fixpoint iteration. The method is presented for first-order programs in flat domains.

[Martin and Hankin, 1987] extends the frontiers algorithm and shows how it can be applied to higher-order functions over arbitrary fine lattices. [Martin, 1989] describes several variants of this and other algorithms for finding fixpoints.

[Hunt, 1989, Hunt and Hankin, 1991] presents a new formulation of the frontiers algorithm based on the insight that frontiers represent open and closed subsets of a function's argument domain. The approach allows the evaluation of approximative fixpoints, generating upper and lower bounds for the true fixpoint.

[Hankin and Hunt, 1992] develops the approach somewhat further and demonstrates the correspondence with the widening/narrowing approach used in traditional abstract interpretation.

[Seward, 1992] extends the Frontiers algorithm to the strictness analysis of polymorphic functions.

Polymorphism

[Abramski, 1985] proves that the result of the strictness analysis of any monomorphic instance of a polymorphic function (but not the abstract function itself) can be used at all instances (*polymorphic invariance*).

[Gebreselassi and Hughes, 1989] defines an abstract interpretation of polymorphic functions which may be used to perform strictness analysis. The abstract version of a polymorphic function is then again polymorphic. However, the method is not able to deal with recursive functions.

[Baraki, 1991] extends this approach to higher-order programs with general functions. The approximations are computed from abstract functions of simple instances of polymorphic functions. In the first-order case, the method gives exact analysis.

[Seward, 1992] extends the frontiers algorithm to the strictness analysis of polymorphic functions avoiding the separate analysis of each monomorphic instance. Therefore this approach is much faster in the presence of polymorphic functions but the approximation is less exact.

Evaluation Transformers

[Burn, 1987b, Burn, 1987a] establish a formal relationship between the *denotational* semantics of a program (derived by abstract interpretation) and its *operational* semantics (implemented by an evaluation mechanism). A criterion is introduced which guarantees that changing the evaluation strategy of a program does not change its semantics. For each point of a strictness domain an *evaluator* is defined which can be safely applied to those objects which are abstracted to this point. A calculus is developed that derives for each function an *evaluation transformer* which takes the evaluator of the result and returns the appropriate evaluators for the arguments.

[Burn, 1988, Burn, 1990a, Burn, 1991a] give compilation schemes that use the evaluation transformer model to generate code for the Spineless G-machine and parallel variants of it. For each function, different versions of the code are generated depending on the amount of evaluation required for the result. In the case of first-order functions, the appropriate version can be selected at compile-time (context-free evaluation transformers); for higher-order functions the appropriate version is selected at runtime taking into account the information available on the argument function.

[Burn, 1991c] provides a formalization of evaluators as reduction strategies in the typed lambda calculus and a proof that using the evaluation transformers obtained from abstract interpretation results in a safe reduction strategy.

[Kuchen and Loogen, 1989, Loogen, 1990] describe the implementation of the evaluation transformer model in a parallel graph reduction machine.

[Howe and Burn, 1992] discusses how to use the results of strictness analysis by evaluation transformers to generate more efficient code for the Spineless Tagless G-Machine.

[Mintchev, 1992] discusses how to apply the results of strictness analysis by evaluation transformers to generate code for a parallel version of the Spineless Tagless G-machine. The notion of *hyper-strictness* is introduced that allows lists to be fully evaluated to normal form.

Contexts and Projections

[Hughes, 1985] bases strictness analysis on the analysis of the *context* in which an expression is evaluated. It is shown how contexts can describe how much information on a data structure is *sufficient* to return a certain amount of the result. However, the calculation of contexts yields equations that have to be solved by algebraic manipulations and heuristic approximations. The approach is restricted to first-order monomorphic languages.

[Hughes, 1987a] provides a formalization of contexts as sets of continuations. However, the model is a not very intuitive and the proofs involved are quite lengthy. [Hughes, 1987b] outlines a higher-order method for strictness analysis with a forward (higher-order) component and a backward (first-order) component based on contexts.

[Wadler and Hughes, 1987] provides a new and more suitable formalization of contexts using the domain theoretic notion of *projections*. Functions are abstracted to projection transformers that map result projections into suitable argument projections. A calculus is developed that derives projection transformers using fixpoint iteration over finite domains. The technique is still restricted to first-order monomorphic programs.

There is a strong relationship between projections and evaluators pointed out by [Wadler and Hughes, 1987] and [Burn, 1987b]. However, the concepts are not identical, since projections are a denotational notion while evaluators are an operational notion. [Burn, 1990c] gives a formal relationship between botch concepts and shows that projections give for first-order programs somewhat more information than abstract interpretation.

[Burn, 1990d] shows how projection analysis can be used for the compilation of functional programs. To each projection, a corresponding evaluator is defined and it is proved that the application of this evaluator is safe in the given context. Compilation rules are given for the Spineless G-Machine that apply the evaluation transformers derived from projection analysis.

A deficiency of the original projection technique is that function arguments are analyzed separately and the derived strictness information is weaker than when the body of a function is inlined into an expression. [Davis and Wadler, 1989] enhances the method by introducing environments of projection transformers where no information on the arguments goes lost.

[Hughes, 1989] uses a categorical framework to extend the projections method to the strictness analysis of first-order polymorphic functions. The drawback of this approach is that analyzing polymorphic functions may give poorer results than analzing each monomorphic instance separately.

[Hughes, 1990] gives an excellent overview on forwards analysis by abstract interpretation and backwards analysis by projections and the relationship between both techniques.

[Davis and Wadler, 1990] describes a relational approach to backward strictness analysis to higher-order monomorphic functions.

[Kubiak *et al.*, 1991] discusses the implementation of first-order, polymorphic backwards strictness analyzer in the Glasgow Haskell Compiler.

Other Approaches

[Wray, 1986] presents an backward analyzer that takes advantage of type information handling a polymorphic type system with higher-order functions. However, it is unable to analyst non-flat domains usefully.

[Tsung-Min and Mishra, 1989] describes an approach to strictness analysis based on nonstandard type inference. The algorithm does not require fixpoint iteration but its results may be weaker since function arguments are analyzed separately.

[Jensen, 1991] presents a framework for comparing strictness analysis by abstract interpretation and strictness analysis by type-inference and proves that both are equally powerful.

[Jones and Le Metayer, 1989] proposes a family of finite domains for the backwards analysis of programs by abstract interpretation. Each domain P_n consists of patterns that describes how much information on the head fields and the tail fields of the first n cells of a list is available. This information can be used to derive for each function by fixpoint iteration the amount of argument information that is necessary to compute a result.

[Pingali, 1990] suggest a backwards strictness analysis method where *logical variables* are used to propagate demand information.

[Dybjer, 1987] presents some preliminary work on an analysis technique called *inverse image analysis* which is able to find for first-order functions information like that available from abstract interpretation. It is not clear whether this method can be automated.

[Chuang and Goldberg, 1991] describes a general method for automatic backward analysis of higher-order languages based on computing inverse images of functions over abstract domains. This method can be also applied to functional programs on non-flat types.

[Hartel *et al.*, 1991] discusses the effects of strictness analysis on the quality of generated code in the compiler for a lazy functional language.

[Nöcker, 1993] presents a strictness analysis approach based on infinite abstract domains whose elements are sets of concrete values. Higher order functions and structured data types can be easily modelled; recursive functions are handled by a technique called reduction base analysis. The analyser is implemented in a compiler for Concurrent Clean.

Weight Analysis

Problem Description

A crucial problem for the parallelization of functional languages is to decide when an expression shall be evaluated by a separate task and when it shall be evaluated locally by the current task. This is not only a problem of when it is *safe* to evaluate an expression remotely (which is addressed by strictness analysis) but in particular when it is *efficient*: since task creation poses a runtime overhead significantly larger than a function call, this overhead only pays if the *weight* of the expression (i.e. the time that its evaluation takes) is sufficiently large. The problem of automatic weight analysis is to annotate each expression e in a program by the number t_e of steps required for the evaluation of e.

Automatic weight analysis is easy for expressions that do not contain calls of recursive functions. In this case, t_e is just the *static size* of e i.e. the number of operations in the expression after all function calls have been expanded. However, weight analysis is very difficult (and in general undecidable) if recursive functions are involved.

In this case, the problem is to derive for each recursive function f a function f_t that describes the number of steps that the execution of f takes for a given argument. The argument of f_t is usually an integer that is an appropriate measure for the *size* of the argument of f. The problem of weight analysis thus reduces to two subproblems:

1. To determine for each program function $f(x)$ an appropriate *size function* $f_s(x_s)$ that determines the size of the result of $f(x)$ and a *time function* $f_t(x_s)$ that determines the time to compute $f(x)$ with respect to the argument size $x_s = s(x)$. This yields a system of recurrence equations involving the f_t and f_s.
2. To derive a *closed form* for f_t that does not contain any (direct or indirect) recurrences anymore. Only such a form can be efficiently manipulated and evaluated in an automatic way.

The second problem is in general undecidable (since it can be reduced to the halting problem). An weight analysis system can therefore only hope to solve special (typically occurring) cases and otherwise has to rely on user support or possibly incorrect heuristics or it returns pessimistic upper bounds for computation times.

An essentially more difficult problem is the weight analysis of *lazy* languages where objects are only evaluated if and when required (demand-driven execution) and where shared substructures are evaluated at most once (fully laziness). In this case also the *context* of an evaluation has to be modelled.

Relevant Literature

Compared to the large amount of work on strictness analysis, there exists surprisingly little literature on the automatic weight analysis of functional programs. This reflects the fact that most implemented systems rely on simple heuristics and runtime support to decide when an expression shall be evaluated in parallel.

[Goldberg, 1988] assumes for the compilation of functional programs into serial combinators that each recursive function has infinite runtime. Furthermore, recursive functions are ordered by the static sizes of the function bodies assuming that all functions have the same recursion depth.

[Young, 1989] discusses termination analysis and operation count count analysis in an untyped functional language. However, both are based on simple abstract interpretation methods and are not effective for recursive functions (the problem of solving recurrences is not touched).

In the context of logic languages, [Tick, 1988] derives the relative weight of predicates in a Guarded Horn Clause program from the static size of the clause body; recursive clauses are assumed to have zero weight. This information is used to assist the runtime scheduler of the system.

The first serious attempt of automatic weight analysis is the the METRIC system [Wegbreit, 1975] analyzing programs in a first-order subset of Lisp without mutually recursive functions. Closed expressions are computed that describe the best-case/worst-case/average computation time of a program and the variance, respectively. The system is fully automatic and works without any user assistance and is in this respect still the most advanced. However, the analysis may also fail and results need not be correct since all conditional tests are assumed to be independent.

[Kasai and others, 1980] describes the TIMER system that generates upper and lower bounds for the complexity of programs written in a small imperative language. The system is semi-automatic in that it requires the user to annotate the program by describing the control variables whose values are a measure for the size of the input and by specifying the rate at which conditional and loop branches are chosen.

[Adachi *et al.*, 1979] develops an algorithm to construct programs that compute time bounds for a class of imperative programs called simple loop programs. [Talcott, 1986] gives a general method to extend a program to a program that returns the computation track of the evaluation. The computation time can be expressed as the length of this track.

The ACE system described by [Le Metayer, 1985, Le Metayer, 1988] generates upper-bound complexity functions for a subset of FP. The system is based on a large library of transformation rules that transform a program to a composition of the complexity function and the size measure. However, the termination and the quality of the transformation depend on the order on which the rules are found in the library.

[Rosendahl, 1986, Rosendahl, 1989] models the weight analysis of first-order functional programs in the framework of abstract interpretation. Each function f is mapped into an abstract version $\bar{f}$ that returns an upper bound for the number of steps that the evaluation of f would take. However, $\bar{f}$ is in general recursive (and may contain calls to program functions) and still requires transformation into a closed form.

[Hickey and Cohen, 1988] describes a system for the average computation time analysis of programs. The analysis is based on *distributions* that describe the probability that program variables take certain values. A performance compiler takes a program in a simple imperative language and an input distribution and generates a system of recurrence equations whose smallest solution is the output distribution. From these equations that are automatically simplified (but not solved) the computation time of the program can be derived. The approach is extended to (but not automated for) recursive FP programs.

[Bjerner and Holmström, 1989] generalizes the results of [Bjerner, 1989] on time analysis in type theory to *lazy* first-order functional programs. An abstract interpretation $\bar{f}(\bar{x}, d)$ of $f(x)$ is constructed that returns the number of steps that the evaluation of $f(x)$ takes where the *demand function* d describes the context in which the result of $f(x)$ is required.

[Wadler, 1988] shows that *projections* can be used for this purpose but points out that the results are rather crude. [Sands, 1989] generalizes this approach to higher-order programs deriving equations for the upper and lower bounds of function weights.

[Debray *et al.*, 1990] describes a system that generates recurrence equations for the computation time of logic programs. Assuming that all predicates are monotonic in their result size and weight, a table-driven method is applied to generate closed upper bounds for the predicate weights. A parallelizing compiler uses this information to generate code that at runtime decides whether to spawn a parallel task.

[Flajolet, 1987] examines the use of generating functions for the average computation time of programs. [Hickey and Cohen, 1988] and [Rosendahl, 1989] contain several references to the relevant literature on solving systems of recurrence equations.

[Waugh, 1991] applies in a parallelizing compiler static analysis to estimate partition size and hence task granularity.

9 Parallelization and Compilation

Problem Description

Compared to the amount of literature on the analysis of functional programs, there are significantly less publications on how to actually utilize the derived information in a parallelizing compiler. We will classify the main approaches in this area as follows:

1. **Parallelizing Lisp:** If Lisp is not restricted to its purely functional subset, a parallelizing compiler for this language has to analyze data dependencies by techniques similar to parallelizing Fortran compilrs in order to generate correct parallel code.
2. **Program Transformation:** A sequence of (source-level) rewriting steps is applied to transform a functional program into an explicitly parallel one. The selection of transformation rules is often guided by the human programmer in an interactive way.
3. **Super Combinators:** For the parallel execution of lazy languages, user-defined functions are usually transformed in combinator definitions that do not access a global environment but only depend on their input parameters. Such "super-combinators" can then be distributed among processors.
4. **Dataflow and Multi-Threaded Execution:** The program is translated into a data flow graph where the individual nodes denote the unit of parallelism. In order to minimize the scheduling overhead, optimizing compilers combine sets of graph nodes into sequentially executed (and therefore more coarse-grain) threads.

Furthermore, several *intermediate languages* have been defined as an architecture-independent compilation target.

One of the first publications in this area is [Moor, 1982] that describes a compiler for the HOPE language that generates code for the parallel graph reduction architecture ALICE.

Parallelizing Lisp

[Marti, 1980] describes a compilation technique for Lisp programs where function arguments without side-effects are evaluated in parallel.

[Boyle *et al.*, 1986] describes a transformational approach to generate shared memory parallel Fortran programs from pure Lisp specifications.

[Harrison III, 1986] investigates the compilation and parallelization of Lisp programs for shared memory multiprocessors. A sequence of transformations is supposed that reveal and exploit parallelism in a Lisp program in the presence of side-effects. Recursive functions are replaced by loops that are distributed among processors.

[Harrison III, 1989] extends this work in a parallelizing Scheme compiler called *Parcel*. Interprocedural analysis is applied to collect information concerning interprocedurally visible side-effects and the lifetimes of dynamically instantiated objects. The analysis is specified as an abstract interpretation of Scheme programs and its correctness is proven. [Harrison III and Ammarguella, 1988] discusses the strengths and weaknesses of the Parcel compiler in more detail.

[Larus and Hilfinger, 1988] describes the technique that the program transformation system *Curare* uses to analyze and restructure Lisp programs for concurrent execution.

Program Transformation

[Johnson, 1985] introduces the technique of *lambda-lifting* for transforming a program with free variables in (local) function definitions into a set of combinator definitions ("supercombinators") that do not access a global environment.

[Bush and Gurd, 1985] discusses the transformation of linear recursion schemes into equivalent double recursive (bilinear) ones and the conversion of bilinear into linear forms. These transformations may be used to control the amount of parallelism in a program.

[Harmer and Boyle, 1989] describes a transformation strategy for Lisp programs that improves the efficiency of functions that generate lists by computing all elements in parallel. The pair-wise synchronization of processes is replaced by a global synchronization, which provides the advantages of conversion to tail-recursive form.

[Darlington *et al.*, 1988, Darlington *et al.*, 1989] describe the HOPE+ Transformation Programming Environment where program transformation techniques are used to interactively transform a functional specification into a parallel program.

[Sharp *et al.*, 1991] presents a methodology for systematically synthesizing algorithms for message passing parallel machines from functional specifications.

[Waugh, 1991] investigates the automatic translation of sequential SML prototypes of vision algorithms into parallel Occam programs. Static analysis is applied to consider argument sizes and flow to estimate partition size and granularity.

Intermediate Languages

[Skedzielewski and Glauert, 1985] defines the intermediate language *IF1* which is a textual description of dataflow graphs and the target of the SISAL compiler. [Skedzielewski and Welcome, 1985] describes how traditional code optimizations can be applied to IF1 code. [Welcome *et al.*, 1986] describes the successor language *IF2* that allows optimizations to be applied in terms of abstract storage.

[Glauert *et al.*, 1988] defines the intermediate language *Dactl* for the implementation of declarative languages on highly parallel machines. It is based on a form on graph rewriting with explicit control of the evaluation mechanism.

[Ariola and Arvind, 1989] defines the intermediate language *P-TAC* (for Parallel Three Address Code) designed to capture the sharing of computations. P-TAC is a minimal functional language where the notion of "locations" is used to model data structure operations.

Super Combinators

[Hudak and Goldberg, 1985b, Hudak and Goldberg, 1985a] introduce the notion of *serial combinators* as the unit of parallelism. A serial combinator is a refinement of a supercombinator where all of its concurrent substructure (arising from independent argument evaluation) is made explicit. It is argued that such serial combinators have in some sense "optimal" granularity.

Based on these ideas, [Goldberg, 1988] describes a parallelizing compiler for a lazy functional language for shared and distributed memory architectures. The compiler applies some heuristics to decompose a program into a set of serial combinators and applies several code optimizations.

[Bloss *et al.*, 1988] outlines several analysis techniques in order to let compilers generate more efficient (sequential and parallel) code.

[Burn, 1990d, Burn, 1991b] describe how the the results of strictness analysis can be applied to generate efficient code based on the evaluation transformer model. Compilation rules are given for sequential machines as well as for shared and distributed memory architectures.

[Kuchen and Loogen, 1989, Loogen, 1990, Hogen *et al.*, 1992] describe a compiler for a distributed graph reduction machine based on the ideas of serial combinators and the evaluation transformer model.

[Hammond, 1989b] presents a parallelizing compiler for PSML, a side-effect free version of SML, into Dactl, a compiler target language based on graph rewriting.

[Sargeant and Watson, 1991] investigates an approach where the program is first compiled with monitoring code inserted and run using smallish data to produce statistics which is then fed back into the compiler. Several heuristics are discussed to exploit this information to generate optimized parallel code

Dataflow and Multithreaded Code

[Sarkar and Hennessy, 1986b] presents a compile-time methodology for *partitioning* a program graph into subgraphs that can efficiently executed in parallel. The partitioning algorithm is driven by costs for execution times and communication sizes and approximates an optimal partitioning.

[Sarkar and Hennessy, 1986a] presents an approach to compile-time *scheduling* intended for applications with predictable run-time behavior. A program is partitioned into as many sequential threads as processors are available; there is no overhead due to runtime scheduling.

[Sarkar, 1989] is a comprehensive source on the issue of partitioning and scheduling of program graphs for parallel execution.

[Oldehoeft and Cann, 1988] describes an automatically parallelizing Sisal compiler for a shared memory parallel architecture.

[Sarkar *et al.*, 1988] discusses another automatically partitioning *Sisal* compiler for shared memory multiprocessors. The compiler uses a set of machine parameters to generate an efficient partitioning of the program for a particular target architecture.

The Ph.D. dissertation [Cann, 1989] describes the design and implementation of several new compiler optimization methods eliminating most of the runtime reference counting and dramatically improving the quality of the generated code.

[Gaudiot and Lee, 1987] describes a compiler that translates Sisal programs into Occam code for transputers. A granularity threshold method is applied to find a process partitioning that minimize the communication costs.

[Böhm and Sargeant, 1989] describes code optimization of the Sisal compiler for the Manchester dataflow architecture.

[Oldehoeft and McGraw, 1990] describes how to mix Sisal code with conventional libraries implemented in Fortran.

[Sarkar and Cann, 1990] describes the design of an automated partitioner for Sisal. [Cann, 1991, Cann, 1992] demonstrate Sisal's performance compared to automatically vectorized and parallelized Fortran code.

[Spier and Szymanski, 1990] discusses interprocess data dependency analysis and optimization for the EPL language.

[Traub, 1986] describes a compiler for the dataflow language *Id* that generates code for the MIT dataflow architecture TTDA. Several optimizations are applied to the intermediate program graph. [Arvind and Nikhil, 1987] shows an example for the code generated from a dataflow program.

[Papadopoulos, 1988] describes the compilation process for the MIT Explicit Token Store Architecture ETS "Monsoon" by translating TTDA instructions into ETS instructions. In contrast to the TTDA, the compiler in advance fixes the rendezvous points of runtime tokens as offsets in the activation frame of the task. Also the translation of imperative code into ETS code is sketched.

[Iannucci, 1988] describes the compilation of dataflow programs into code for a hybrid dataflow/von Neumann architecture. In contrast to pure dataflow models, the compiler generates "scheduling quanta" i.e. sequential threads consisting of a few instructions.

[Milewski and Mori, 1990] discusses the efficient implementation of loops and streams in the dataflow language Id.

[Traub, 1989, Traub, 1991a] present a new view on the compilation of lenient (i.e. non-strict but not necessarily lazy) functional languages. The view is based on *partitioning* a function into a set of sequential threads within which the order of statements is fixed at compile time. The partitioning is separated from the decision how threads are scheduled and communicate with each other. If a thread requires the result of another thread, it executes a *fork* statement to derive this value.

On a single-processor machine, *fork* may explicitly switch control from the requesting thread to the thread computing the value. On a multi-processor, *spark* statements may actually trigger the concurrent execution of threads before the *fork* is done. Then *fork* merely has to investigate the contents of a tagged location to retrieve the value.

[Schauser *et al.*, 1991] presents a compiler that partitions a program into sequential threads for an abstract multi-threaded architecture. *Dual dataflow graphs* are used as a program representation where control flow and dataflow are denoted by separate arcs. This increases the possibilities for reducing redundant controlflow and merging partitions. Threads are then compiled to native machine code. The system is implemented for several parallel machines.

[Traub, 1991b] compares the multi-threaded model of code generation with the code generator applied in the "Monsoon" dataflow architecture.

[Loidl, 1992a, Loidl, 1992b] describe a functional language compiler that generates code for a hybrid dataflow/von Neumann architecture implemented on a multi-transputer system.

[Roh *et al.*, 1993] discusses the generation of "dataflow clusters" i.e. coarse-grained dataflow graphs from fine-grained ones and presents extensive simulation results.

[Ang, 1993] presents a new approach to the efficient compilation of sequential loops in dataflow programs.

10 Abstract Machines and Runtime Systems

Problem Description

For parallel implementations of functional languages, runtime support is usually considerably more complex than for imperative languages: a garbage-collected global heap must be managed and (especially for languages with non-strict semantics) an efficient task management with fast context switching is required.

Several new architectures based on graph reduction or dataflow have been proposed to fulfill these demands. However, only few of these proposals have actually been realized in hardware, while most have been simulated on conventional multi-processors. In most papers, the term "machine" therefore actually denotes "abstract machine" (= interpreter) rather than "concrete machine" (= hardware); the term "implementation" is usually understood as "interpretation" or "simulation".

In order to implement their execution more efficiently (in hardware or software), there is a strong trend to make these abstract machines closer to conventional architectures, while more and more effort is put into the compiler that generates the abstract machine code. The abstract machine therefore gradually develops into a runtime system that only supports those features that are not directly provided by the underlying hardware.

The main proposals in this area are based on the principles of

1. Parallel graph reduction or term rewriting,
2. Parallel programmed graph reduction,
3. Dataflow and multi-threaded execution.

In most early implementations of lazy languages, the program graph was interpreted at run-time and reduced to the result in normal form. Programmed graph reduction is an enhancement of this principle, were functions are compiled into machine code whose (non-interpretative) execution updates the program graph.

In dataflow machines, every program expression is a separate task that is started when its input data become available; multi-threaded models are a combination of dataflow with the von-Neumann principle: the compiler generates a set of threads within which instructions are executed sequentially without blocking; these threads are scheduled at runtime by the availability of data.

As emphasized in the introduction, we do not attempt to list all proposals for new parallel architectures for functional programming. Instead we try to sketch the evolution described above by several representative examples.

LISP

[Feeley and Miller, 1989] describes a parallel virtual machine for the compilation of Scheme programs. The virtual machine supports parallel computation based on the `future` construct and is the basis of the *Gambit* compiler.

Graph Reduction and Term Rewriting

[Darlington and Reeve, 1981] describes the parallel graph reduction machine *Alice* that has become the origin of many later graph reduction machines, both abstract and in hardware. The execution model of this machine is based on interpreted (non-programmed) graph reduction and has been implemented on multi-transputer systems.

[Kluge, 1983] describes a system of cooperating reduction machines that implements in a distributed way a full lambda calculus.

[Barendregt *et al.*, 1987a] gives an operational description of parallel graph rewriting and shows its soundness and completeness with respect to term rewriting. The companion paper [Barendregt *et al.*, 1987b] defines the experimental language *Lean* that specifies computations by sets of rewrite rules. *Lean* also allows to express imperative constructs, non-determinism and parallelism and is intended as an intermediate language between functional languages and parallel architectures.

[Watson, 1986] describes an approach to the parallel evaluation of functional languages based on the lambda calculus. [Nielson, 1989] extends the lambda calculus by the notion of processes as first class citizens in the style of CCS.

[Zimmer, 1991b, Zimmer, 1991a] discuss the abstract reduction machine π-RED that is based on the syntactic reduction of program expressions. The system may incorporate imperative subsystems while preserving the Church-Rosser property. [Buelck and others, 1993] describes the implementation of this system on a distributed memory hypercube.

Programmed Graph Reduction

[Lemaitre *et al.*, 1986] sketches the design of a parallel reduction machine based on a (fixed) combinator machine language.

[Peyton Jones, 1987] describes the implementation of functional languages by programmed graph reduction on the basis of the G-machine and also sketches some parallelization issues.

[Peyton Jones, 1989] surveys several attempts to parallel implementations of functional languages based on the concept of parallel graph reduction.

[Augustsson and Johnsson, 1989, Johnsson, 1991] introduce the $\langle v, G\rangle$-*machine*, a parallel extension of the G-machine implemented on a shared memory multi-processor.

[George, 1989] describes another parallel variant of the G-machine for shared memory multi-processors. Special consideration is given to minimize the runtime overhead of the abstract machine.

[Maranget, 1991] describe another shared memory implementation of Lazy ML based on a parallel a parallel extension of the G-machine. Parallelism is introduced by *fork* annotations by the programmer.

[Milikowski and Vree, 1991] describes the G-line, a distributed memory variant of the G-machine. By horizontal micro-code, micro-parallelism is exploited for the reduction of a single graph node.

[Peyton Jones and Lester, 1992] contains the executable code of an abstract parallel G-machine and a compiler for it.

[Burn, 1988] describes a shared memory variant of the G-machine based on the evaluation transformer model. [Bevan *et al.*, 1989] and [Burn, 1991b] extends this work to distributed memory architectures. [Lester and Burn, 1989, Burn, 1991a] define the *HDG* (Highly Distributed Graph Reduction) machine.

[Kuchen and Loogen, 1989, Loogen, 1990] describe the distributed graph reduction machine *PAM* that is based on the evaluation transformer model.

[Goldberg and Hudak, 1987] describe the *Alfala* system, a distributed memory implementation of a lazy functional language on a hypercube multiprocessor where the abstract machine model is based on "heterogeneous graph reduction" a hybrid of graph reduction and stack oriented execution. [Goldberg, 1988] describes this system and its shared memory variant *Buckwheat* in detail.

[Nöcker *et al.*, 1991b, Kesseler, 1991] describe the Parallel ABC Machine for the implementation of Concurrent Clean on distributed memory architectures.

[Rabhi, 1991] describes experiments with a programmed graph reduction architecture implemented on a transputer system.

[Peyton Jones and Salkild, 1989, Peyton Jones, 1991, Peyton Jones, 1992] describe the *spineless tagless G-machine*, an evolution of the original G-machine. A program is compiled into the *STG language* which is the (rather high-level) abstract machine code for the machine. The efficient mapping of this abstract machine on sequential and parallel stock hardware by translation into C code is discussed.

[Hwang and Rushall, 1992] describes a parallel version of the Spineless Tagless G-machine for a distributed memory architecture. Expressions marked by parallelization annotations are evaluated in parallel to weak head normal form. An operational semantics of the abstract machine is defined and a simulator for the machine is described.

[Hogen and Loogen, 1992b, Hogen and Loogen, 1992a] describe another organization of task stacks for parallel graph reduction machines. Instead of allocating a heap node for each function invocation, a scheme is proposed to allocate the workspace of each task on a single *meshed stack* which is organized as a contiguous array but shared by several processes.

Dataflow and Multi-Threaded Execution

[Arvind and Nikhil, 1987] describes the *MIT Tagged Token Data-flow Architecture (TTDA)* that has become the prototype for many other dataflow projects. Programs in the functional language ID are compiled into dynamic dataflow graphs that are the machine language of the TTDA. Special support is given to the management of non-strict data structures, the so called *I-Structures*.

[Papadopoulos, 1988, Papadopoulos and Culler, 1990] describes the successor of the TTDA, the *MIT Explicit Token Store Architecture (ETS) "Monsoon"*. In contrast to the TTDA, were special hardware is required to match tokens at runtime, the rendezvous points of tokens is determined at compile-time as a fixed location in the activation frame of a task.

[Iannucci, 1988] defines an hybrid architecture that combines features of conventional architectures with those of dataflow architectures. In contrast to the TTDA, where the scheduling of instructions takes place entirely at runtime, a compiler generates for this hybrid architecture scheduling quanta within which the order of instructions is fixed at compile-time.

[Nikhil, 1989] proposes a architecture that unifies dataflow machines with dataflow architectures introducing the idea of sequential threads into a dataflow context. Issues of synchronization and instruction scheduling are separated from those of local memory access.

[Schreiner, 1990a, Schreiner, 1990b, Schreiner, 1991] define an abstract dataflow machine *ADAM* which is based on another extension of the von-Neumann principle. Special fork and synchronization instructions are added to an otherwise conventional machine language such that dataflow behavior can be simulated.

[Ostheimer, 1991] describes the abstract *STAR:DUST* architecture which is a hybrid of dataflow and von-Neumann architectures for the efficient execution of small sequential threads.

[Culler *et al.*, 1991] defines an abstract multi-threaded machine TAM which is much closer to conventional von Neumann architectures than to dataflow architectures. Within a function's code-block several threads may exist executing sequences of instructions. Communication between threads is done via conventional registers, synchronization only occurs at the start of a thread. The compiler has explicit control over the scheduling of quanta i.e. sets of threads within an activation.

11 Dynamic Program Control

Problem Description

In this section, we focus on those publications that in particular deal with runtime aspects of functional programs i.e. with

1. **Simulation:** Several systems have been especially designed to simulate the execution of functional programs on (perhaps not yet existing) parallel architectures in order to get information about the dynamic behavior of the program under different scheduling strategies.
2. **Parallelism Control:** Since functional programs often contain a lot of inherent parallelism but a parallel machine has only finite (in particular memory) resources, the amount of parallelism must be carefully controlled without sacrificing the utilization of processors.
3. **Task Scheduling:** Especially on loosely coupled (distributed memory) machines, the mapping of tasks to processors can have an tremendous impact on the performance of the system. Strategies have to be found that minimize the runtime (in particular communication) overhead but keep all processors busy.

All aspects are closely tied together, e.g. by decreasing the amount of parallelism, the balancing of load becomes more difficult.

Simulation

[Keller and Lin, 1984] describes a simulator for the parallel graph reduction architecture *Rediflow*.

[Nikhil, 1987c] describes *Id World* an integrated dataflow simulation and programming environment that allows to measure many interesting aspects of parallelism (like parallelism profile, memory consumption, and speedup).

[Weening, 1988] describes the parallel Lisp simulator *CSIM* that models a shared memory multiprocessor executing Common Lisp programs.

[Deschner, 1989] describes a system that simulates the parallel execution of functional programs based on compiled graph-reduction.

[Watson, 1989] describes a simulator for the architecture of the European Declarative System (EDS).

[Bennett and Kelly, 1992] describes a scheme for the event-driven simulation of parallel graph reduction on a shared memory multiprocessor. The simulator allows to monitor the sharing of heap cells and the effect of various process management strategies using a variety of cache designs.

[Dörr, 1992] presents a program monitoring method which reduces the costs of event tracing by using a model to manage the system's information which has been collected since the beginning of the run.

Parallelism Control

[Syre, 1976] discusses the control of parallelism and synchronization in a single assignment language.

[Burton, 1985a, Burton, 1985b] address the requirements of speculative computation i.e. controlling computation and reclaiming unnecessary computation. A parallel construct similar to the `future` is suggested which has an additional priority argument used for the scheduling of tasks.

[Arvind and Culler, 1987] describes *loop bounding* as a technique to control the amount of parallelism in dataflow programs and gives several simulation results.

[Arvind *et al.*, 1988a] uses simulation results to assess the benefits of fine-grain parallelism in dataflow programs. The effects of increasing the granularity of dataflow programs is discussed.

[Ruggerio and Sargeant, 1987] discusses the control of parallelism in the Manchester Dataflow Machine. Several software mechanisms for parallelism control are investigated and found inadequate and an architectural mechanism named *throttle* is introduced. Simulation results demonstrate its effectiveness.

[Peyton Jones *et al.*, 1989] discusses static and dynamic techniques to control the growth of parallelism in the graph reduction architecture GRIP. Dynamic techniques are applied to hold data near to the processors that references it.

[Osborne, 1988] suggests the use of *sponsors* to control speculative parallelism in Multilisp. The allocation of resources to tasks is controlled by task attributes that are supplied by sponsor tasks.

[Rabhi and Manson, 1991, Rabhi, 1992] examine the granularity control of functional programs (in particular based on the "divide and conquer" paradigm).

[Bratvold, 1993, Michaelson *et al.*, 1993] use profiling information and performance models for skeletons in their *SkelML* compiler to control the granularity of programs.

Task Scheduling

[Hudak, 1983, Hudak and Goldberg, 1984, Hudak and Goldberg, 1985a] explore a model of *diffused combinator reduction* in which the reduction process is distributed by demand among a network of closely-coupled processors. The ideas are tested by simulation and experimental results are given.

[Schmittgen *et al.*, 1986] discusses a workload balancing scheme for a system of cooperating reduction machines.

[Goldberg, 1988] investigates several task distribution and load balancing schemes in the distributed memory implementation *Alphala* and the shared memory implementation *Buckwheat* of the lazy language Alfl. A large number of experiments for a variety of programs is described. A summary of these results can be also found in [Goldberg and Hudak, 1987].

[Halstead, 1988] discusses several strategies for task creation and task scheduling in the implementation of Multilisp. The issue of speculative computation is addressed and scheduling strategies of different Multilisp implementations are reviewed.

[Mohr *et al.*, 1990] describes a *lazy task creation* technique to increase the granularity of functional programs.

[Hammond and Peyton Jones, 1991] presents empirical results of several runtime strategies for the creation and scheduling of tasks in the GRIP architecture. It turns out that the load management system is not responding sufficiently rapidly to changes in the system load.

[Langendoen, 1991] investigates for a distributed graph reduction machine consisting of a set of processor clusters the opportunities to exploit the shared memory of a cluster and its associated caches.

[Aharoni *et al.*, 1991] presents a dynamic, adaptive mechanism to control the granularity of parallel functional programs. It is based on the ability to expose at run time the shape of the execution structure and to allocate the processors accordingly. Continuous sampling allows to adapt to structure changes.

[Pehoushek and Weening, 1988] describes techniques for low-cost process creation and dynamic partitioning of Qlisp programs and gives several experimental results.

[Achten, 1991] discusses several load distribution strategies using the PABC simulator for Concurrent Clean.

[Glas *et al.*, 1992] describes the implementation of parallel branch-and-bound algorithms in a functional language. It discusses several strategies and the trade-off between a lower average processor utilization as a result of synchronous communication and the performance loss caused by exploitation of speculative parallelism.

[Achten, 1991] presents the results of several experiments on a prototype runtime system for Concurrent Clean. Based on simulation, several synchronization and communication strategies are described.

12 Implementations

Problem Description

We will conclude this bibliography with a list of parallel functional languages

> *that have been implemented on conventional parallel architectures.*

Concerning the rather large amount of literature on parallel functional programming, one would expect that there are also many operational (at least prototype) systems around. However, several languages have been implemented only on graph-reduction or dataflow architectures, and many implemented systems are actually single-processor simulations.

The following list only includes literature that describes implementations of functional languages on conventional parallel hardware (also simulations on workstation networks). In case the most recent publication of a project describes still ongoing work, we use the term "probably implemented".

A final note about the the efficiency of these implementations. Many published experiments only present *relative speedups* of parallel functional programs compared to the same program executed on a single processor. Few publications talk about the *absolute performance* of a program compared to (sequential or explicitly parallel) imperative code.

At present, the situation seems as follows: Several explicitly parallel (or para-functional) dialects of Lisp and ML are very efficient, and also automatically parallelizing Lisp compilers seem to produce qualitatively good code. The data parallel Connection Machine Lisp is efficient but restricted to SIMD.

Among purely functional languages, only the implementations of SISAL (and perhaps also of Id and other dataflow languages) are able to compete with explicitly parallel imperative code. All parallel implementations of lazy languages suffer from the sequential execution overhead (which is not totally removed by strictness analysis) as well as the overhead introduced by complicated parallel runtime systems.

Efficiency is therefore still the main problem in parallel functional programming. People using parallel computers are interested in performance, not in relative speedups of slow sequential programs. If these people shall be convinced that "functional languages are the choice for parallel programming", this problem has to be addressed more adequately.

Parallelizing Lisp Compilers

The parallelizing Lisp compiler *Curare* [Larus and Hilfinger, 1988] and the *Parcel* compiler [Harrison III, 1986, Harrison III, 1989] generate code for shared memory multiprocessors.

Explicitly Parallel Languages

[Brandenburg *et al.*, 1988] describes the implementation of a parallel *Common Lisp* variant for the Intel iPSC/2 hypercube.

ZLISP [Dimitrovsky, 1988] has been implemented on the Ultracomputer prototype at the New York University.

A prototype of *dLisp* [Weber, 1989] has been implemented on a local area network of workstations.

Spur Lisp [Zorn *et al.*, 1987, Zorn *et al.*, 1988, Zorn *et al.*, 1989] has been implemented on the Spur multiprocessor workstation developed at the University of California at Berkeley.

EDS Lisp [Hammer and Henties, 1990, Ilmberger Hermann, 1991] has been implemented on the European Declarative System prototype computer.

Tlisp [Koorey Willcock, 1992] has been implemented on a multi-transputer system.

The *explicitly parallel ML* extension described by [Cooper and Morrisett, 1990, Morrisett and Tolmach, 1992] has been implemented on several shared memory multiprocessors and workstations.

[Matthews, 1991] describes a distributed implementation of Standard ML.

FP2 [Jorrand, 1984, Jorrand *et al.*, 1985, Kurfeß *et al.*, 1990] has probably been implemented on multi-transputer systems and on the Stollman Test Machine (STM) emulating a dataflow system.

Erlang [Armstrong and Virding, 1991, Armstrong *et al.*, 1991, Armstrong *et al.*, 1992, Armstrong *et al.*, 1993] has been implemented on special-purpose telephony switching hardware and on workstations.

Data-Parallel Languages

Connection Machine Lisp [Steele Jr. and Hillis, 1986] has been implemented on the Thinking Machines Connection Machine 2 SIMD computer.

An SIMD variant of *EuLisp* [Merrall, 1991b, Merrall, 1991a, Padget and Merrall, 1991] has been probably implemented on a Maspar MP-1 SIMD machine.

The nested data-parallel language *NESL* [Blelloch, 1993, Blelloch *et al.*, 1993] runs on several vector processors and SIMD machines.

Functional Parallel Languages

Multilisp [Halstead, 1984, Halstead, Jr., 1985, Halstead, Jr., 1986, Halstead, 1986, Halstead, 1987] has been implemented by a byte code interpreter on the MIT Concert multiprocessor.

MultiScheme [Miller, 1987, Miller, 1988] has been implemented by an interpreter on the BBN Butterfly machine. A parallel version of *Portable Standard Lisp* [Swanson *et al.*, 1988] has also been implemented for this architecture.

The Multilisp variant *Mul-T* [Kranz *et al.*, 1989, Kranz *et al.*, 1988] has been implemented by an optimizing compiler on the Encore Multimax shared memory multiprocessor.

Qlisp [Gabriel and McCarthy, 1984, Gabriel and McCarthy, 1988, Goldman and Gabriel, 1988b, Goldman and Gabriel, 1988a, Goldman and Gabriel, 1989, Goldman *et al.*, 1988] has been implemented on an Alliant FX/8 shared memory multiprocessor.

Concurrent Scheme [Kessler and Swanson, 1988] has been implemented on a BBN GP1000 shared memory multiprocessor, on a workstation network and on the experimental Mayfly distributed memory multiprocessor developed by the Hewlett-Packard Laboratories.

Para-Functional Languages

A para-functional variant *ParAlfl* [Hudak, 1986c, Hudak and Smith, 1986] of the lazy language Alfl has been implemented on top of *Alfala* system for Intel iPSC/2 hypercubes as well as the *Buckwheat* system for Encore Multimax shared memory machines.

The $\langle v, G\rangle$*-machine* [Augustsson and Johnsson, 1989] for a para-functional version of *Lazy ML* has been implemented on a Sequent Symmetry shared memory multiprocessor.

GAML [Maranget, 1991], another para-functional implementation of Lazy ML, has been implemented on the Sequent Balance shared memory multiprocessor.

The skeleton compiler for *SkelML* [Bratvold, 1993, Michaelson *et al.*, 1993] generates Occam2 code for a multi-transputer system.

Lazy Languages

Alfl has been implemented on an Intel iPSC/2 hypercube system (*Alfala*) and on a Encore Multimax shared memory multiprocessor (*Buckwheat*) [Goldberg and Hudak, 1987, Goldberg, 1988]. Also a runtime system for a CRAY multiprocessor has been developed [Michelson *et al.*, 1986].

The *HDG* (Highly Distributed Graph Reduction Machine) [Burn, 1991a], a parallel version of the G-machine based on the evaluation transformer model, has been implemented on a transputer network.

The parallel graph reduction architecture *PAM* [Kuchen and Loogen, 1989, Loogen *et al.*, 1989, Loogen, 1990] which is also based on the evaluation transformer model has been implemented on a multi-transputer system.

The graph reducer described in [Rabhi and Manson, 1991] has been implemented on a multi-transputer system.

[Buelck and others, 1993] describes the implementation of the Π-*Red* graph reduction system on a distributed memory hypercube.

Dataflow Languages

The single assignment language *Lucid* [Ashcroft and Wadge, 1977, Wadge and Ashcroft, 1985, Ashcroft *et al.*, 1991] has been implemented on several multiprocessor systems.

Likewise the dataflow language *Crystal* [Chen, 1985, Chen *et al.*, 1991] has been implemented on several multiprocessors including hypercube systems.

The equational programming language *EPL* [Szymanski, 1991a, Szymanki, 1993] has been compiled for several SIMD architectures, a shared memory computer and message passing systems.

The single assignment language *SISAL* has been implemented on a variety of dataflow computers [Allan and Oldehoeft, 1985], vector-supercomputers [Lee *et al.*, 1988], systolic arrays [Gross and Sussman, 1987], shared memory [Oldehoeft and Cann, 1988, Sarkar *et al.*, 1988, Garsden and Wendelborn, 1990] and distributed memory multiprocessors [Evripidou *et al.*, 1989, Grit, 1990]. At present these implementations seem to be the most efficient parallel implementations of functional languages and can compete with imperative programs.

The dataflow language *Id* has been implemented by a simulator on a network of workstations [Nikhil, 1987c] and on the ETS Monsoon dataflow computer [Papadopoulos, 1988, Papadopoulos and Culler, 1990, Traub, 1991b]. An optimizing compiler also generates code for the abstract multithreaded machine *TAM* [Culler *et al.*, 1991] that is translated into code for a variety of sequential and parallel architectures.

The abstract dataflow machine *ADAM* [Schreiner, 1990a, Schreiner, 1991] programmed in the functional language EVE [Loidl, 1992a] has been implemented on a multi-transputer system.

Bibliography

[Abramski and Sykes, 1985]
S. Abramski and R. Sykes.
Secd-m: A Virtual Machine for Applicative Programming.
In Jean-Pierre Jouannaud, editor, *FPCA '85, Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 81–98, Nancy, France, September 16–19, 1985. Springer, Berlin.

Abstract: We present a virtual machine to support *applicative multiprogramming* — the description of concurrent, asynchronous systems such as operating systems in a functional style. The machine extend's Landin's secd machine to support multiple concurrent expression evaluation, non-determinism in the form of the fair merge, and a full range of input and output devices. This allows system programs to be written in a functional style. The secd-m machine has been implemented and a number of functional concurrent programs demonstrated.

Keywords: SECD, Abstract Machine, Operating System.

[Abramski, 1985]
Samson Abramski.
Strictness Analysis and Polymorphic Invariance (Extended Abstract).
In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects, Proceedings of a Workshop*, volume 217 of *Lecture Notes in Computer Science*, pages 1–23, Copenhagen, Denmark, October 17–19, 1985. Springer, Berlin.

Abstract: In this paper an approach to strictness analysis for polymorphic functions is developed.

Keywords: Strictness Analysis, Abstract Interpretation.

[Abramsky and Hankin, 1987]
S. Abramsky and C. Hankin, editors.
Abstract Interpretation of Declarative Languages.
Ellis Horwood, 1987
Keywords: Abstract Interpretation.

[Achten, 1991]
Peter Achten.
Annotations for Load Distribution.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 247–264, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: Parallel programming is the art of gaining vast computational speed-up by finding a well (or optimal) load distribution of a program onto some network of processors. Optimal load distribution of programparts on a network of processors is an undecidable problem. Therefore supplying the programmer with annotations for directing the actual load distribution of a certain program still seems to be a necessary method of increasing the programs performance. The talk discusses what a good set of primitives to describe load distributions in Concurrent Clean might look like. Some of the primitives have been implemented on the PABC simulator, on which a number of parallelized programs have been tested.

Keywords: Para-Functional Programming, Annotations, Process Mapping.

[Adachi *et al.*, 1979]
Akeo Adachi, Takumi Kasai, and Etsuro Moriya.
A Theoretical Study on the Time Analysis of Programs.
Number 74 in Lecture Notes in Computer Science, Springer, Berlin, 1979.

Abstract: An algorithm is developed that constructs time bound programs for a class of imperative programs called simple loop programs.

Keywords: Granularity Analysis.

[Aharoni *et al.*, 1991]
Gad Aharoni, Yaron Farber, and Amnon Barak.
A Strategy for the Run-Time Mangagement of Fine-Grained Parallelism.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 227–246, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: We present a dynamic, adaptive mechanism for granularity control of the parallel execution of functional programs. This mechanism utilizes the inherent parallelism of functional programs, yet it endeavors to limit excessive parallelism. It is based on the ability to expose, at run time, the shape of the underlying execution structure of the program, and then to allocate the tasks to the processing elements accordingly. High-density structures are distributed quickly among many processing elements, whereas low density ones have a tendency to be confined to few processors. Continuous sampling of the execution structure allows self-tuning of the mechanism, when the underlying structure changes gradually. The advantage of the presented mechanism is its ability to handle programs whose execution structure cannot be determined in advance. We present performance figures that show the effectiveness of the approach for a wide range of computation structures, including divide-and-conquer programs, and linear recursive algorithms.

Keywords: Load Balancing, Granularity Control.

[Akerholt *et al.*, 1991]
G. Akerholt, K. Hammond, S. Peyton Jones, and Ph. Trinder.
A Parallel Functional Database for GRIP.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 7–30, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: GRIP is a shared memory multiprocessor designed for efficient parallel evaluation of functional languages using compiled graph reduction. In this paper, we consider the feasibility of implementing a data base manager on GRIP, and present results obtained from a pilot implementation. A database implemented in a pure functional language must be modified *non-destructively*, i.e. the original database must be preserved and a new copy constructed. The naive implementation provides evidence for the feasibility of a pure functional database in the form of modest real-time speed-ups, and acceptable real-time performance. The functional database is also used to investigate the GRIP architecture, compared with an idealized machine. The particular features investigated are the thread-creation costs and caching of GRIP's distributed memory.

Keywords: Databases.

[Allan and Oldehoeft, 1985]
S. J. Allan and R. R. Oldehoeft.
HEP SISAL: Parallel Functional Programming.
In J. S. Kowalik, editor, *Parrallel MIMD Computation*, pages 123–150. MIT Press, Cambridge, MA, 1985
Keywords: Dataflow, Sisal.

[Allison, 1987]
Lloyd Allison.
Two Functional Programming Techniques: Continuations, Circular Programs.
Technical Report 87/91, Department of Computer Science, Monash University, Clayton, Australia, January 1987.

Abstract: Continuations are used in denotational semantics to describe control commands such as jumps. Here it is shown how they can be used as a programming technique to simulate backtracking and coroutines. A *circular program* creates a data structure where the computation of new components depends on existing components. Examples arise where a list is to be created in which later elements depend on earlier ones. Such programs can avoid creating garbage in the form of intermediate lists, bringing some of the efficiency of imperative programming to functional programming. The technique can sometimes be used in an imperative language to create an elegant program in a functional style.

Keywords: Continuation, Backtracking, Coroutine, Non-Determinism.

[Allison, 1989a]
Lloyd Allison.
Applications of Recursively Defined Data Structures.
Technical Report 88/119, Department of Computer Science, Monash University, Clayton, Australia, February 1989.

Abstract: A *circular program* contains a data structure whose definition is self-referential or recursive. Use of such definition allows efficient functional programs to be written that would not otherwise be possible. A novel primes program and applications to the Fibonacci numbers, to Ackermann's function and to constraint-satisfaction problems are given.

Keywords: Circular Programs, Streams.

[Allison, 1989b]
Lloyd Allison.
Circular Programs and Self-referential Structures.
Software — Practice and Experience, 19(2):99–109, 1989.

Abstract: A *circular program* creates a data structure whose computation depends upon itself or refers to itself. The technique is used to omplement the classic data structures circular and doubly-linked lists, threaded trees and queues, in a functional programming language. These structures are normally thought to require updateable variables found in imperative languages. For example, a functional program to perform the breadth-first traversal of a tree is given. Some of the examples result in circular data structures when evaluated. Some examples are particularly space-efficient by avoiding the creation of intermediate temporary structures which would otherwise later become garbage. Lastly, the technique can be applied in an imperative language to give an elegant program.

Keywords: Circular Program.

[Anderson and Hudak, 1990]
Steven Anderson and Paul Hudak.
Compilation of Haskell Array Comprehensions for Scientific Computing.
In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25(6) of *SIGPLAN NOTICES*, pages 137–149, White Plains, New York, June 20–22, 1990.

Abstract: Monolithic approaches to functional language arrays, such as Haskell *array comprehensions*, define elements all at once, at the time the array is created, instead of incrementally. Although monolithic arrays are elegant, a naive implementation can be very inefficient. For example, if a compiler does not know whether an element has zero or many definitions, it must compile runtime tests. If a compiler does not know inter-element data dependences, it must resort to pessimistic strategies such as compiling elements as thunks, or making unnecessary copies when updating an array. Subscript analysis, originally developed for imperative language vectorizing and parallelizing compilers, can be adapted to provide a functional language compiler with the information needed for efficient compilation of monolithic arrays. Our contribution is to develop the number-theoretic basis of subscript analysis with assumptions appropriate to functional arrays, detail the kinds of dependence information subscript analysis can uncover, and apply that dependence information to sequential efficient compilation of functional arrays.

Keywords: Arrays, Updating Aggregates.

[Ang, 1993]
Boon Seong Ang.
Efficient Implementation of Sequential Loops in Dataflow Computation.
In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 169–178, Copenhagen, Denmark, June 9–11, 1993. ACM Press.

Abstract: The implementation of sequential loops in dataflow computation had traditionally not received very much attention as it was assumed that most loops would be executed in parallel. This assumption was valid for earlier dataflow machines such as the MIT Tagged Token Dataflow Architecture (TTDA), Sigma-1 but not for the newest generation of dataflow machines including Monsoon, EM-4 and Epsilon-2. On the latter machines, sequential loops use less memory, and can execute in fewer instructions, albeit with lower parallelism than the parallel versions. This characterisation of sequential and parallel loops suggests that programs should have parallel outer loops and sequential inner loops. The run time of sequential loops therefore becomes significant in the overall run time. We also found that previous implementations of sequential loops can incur fairly high overheads. In this paper, we present two new ways of implementing sequential loops that have lower overhead than previous methods. We studied this problem in the context of compiling Id for Mosoon.

Keywords: Dataflow, Compilation, Granularity.

[Ariola and Arvind, 1989]
Zena Ariola and Arvind.
P-TAC: A Parallel Intermediate Language.
In *FPCA '89, Functional Programming Languages and Computer Architecture*, pages 230–242, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: P-TAC is an intermediate-level language designed to capture the sharing of computation. It is a more suitable internal language for functional language compilers than the *λ-calculus or combinators*, especially for compiler optimizations. Using P-TAC, a proof for the confluence of Id, a higher-order functional language augmented with I-structures, is given. Using the notion of observational congruence the correctness of some compiler optimizations is shown.

Keywords: Compilation, Target Language.

[Armstrong and Virding, 1991]
J. L. Armstrong and R. Virding.
Erlang — An Experimental Telephony Switching Language.
In *XIII International Switching Symposium*, Stockholm, Sweden, May 27 – June 1, 1991.

Abstract: We present an experimental programming language called Erlang which is suitable for programming telephony applications. We discuss some of the requirements of such a language and introduce it by a series of small examples which show how both of sequential and concurrent activities can be programmed. We discuss the error recovery mechanism used in Erlang and the performance characteristics of the current implementation.

Keywords: Erlang, Message Passing.

[Armstrong *et al.*, 1991]
J. L. Armstrong, R. H. Virding, and M. C. Williams.
Erlang User's Guide and Reference Manual Version 3.2.
Technical report, Ellemtel Utvecklings AB, Sweden, 1991
Keywords: Erlang, Message Passing.

[Armstrong *et al.*, 1992]
J. L. Armstrong, B. O. Daecker, S. R. Virding, and M. C. Williams.
Implementing a functional language for highly parallel real-time applications.
In *Software Engineering for Telecommunication Switching Systems*, Florence, Italy, March 30 – April 1, 1992.

Abstract: This paper describes a fast, highly portable implementation of the real time, symbolic, declarative programming language Erlang.

Keywords: Erlang, Message Passing, Implementation.

[Armstrong *et al.*, 1993]
J. Armstrong, D. Williams, and R. Virding.
Concurrent Programming in Erlang.
Prentice Hall, 1993.

Abstract: Erlang is a concurrent functional programming language designed for large industrial real-time systems. Erlang is dynamically typed and has a pattern matching syntax. Functions are defined using recursion equations. Erlang provides explicit concurrency, has asynchronous message passing and is relatively free from side effects.

Keywords: Erlang, Message Passing.

[Arvind and Brock, 1984]
Arvind and J. D. Brock.
Resource Managers in Functional Programming.
Journal of Parallel and Distributed Computing, 1:5–21, 1984.

Abstract: By incorporating the history of communication into a stream, functional programs can be written for I/O and communication. Using the stream concept, managers may be written to control access to resources shared by several processes.

Keywords: Streams, Process Networks.

[Arvind and Culler, 1987]
Arvind and David E. Culler.
Resource Requirements of Dataflow Programs.
Computation Structures Group Memo 280, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December 1987.

Abstract: Parallel execution of programs requires more resources and more complex resource management than sequential execution. If concurrent tasks can be spawned dynamically, programs may require an inordinate amount of resources when the potential parallelism in the program is much greater than the amount of parallelism the machine can exploit. We describe *loop bounding*, a technique for dynamically controlling the amount of parallelism exposed in dataflow programs. The effectiveness of the technique in reducing token storage requirements is supported by experimental data in the form of *parallelism profiles* and *waiting-token profiles*. Comparisons are made throughout with more conventional approaches to parallel computing.

Keywords: Dataflow, Resource Allocation, Parallelism Profiles.

[Arvind and Ekanadham, 1987]
Arvind and Kattamuri Ekanadham.
Future Scientific Programming on Parallel Machines.
Computation Structures Group Memo 272, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1987.

Abstract: A language for large scientific applications should facilitate encoding and debugging of programs at the highest level possible. At the same time it should facilitate generation of efficient code for parallel machines. Often these two requirements are conflicting, and trade-offs must be made. Functional and other declarative languages offer relief on both counts. The use of higher-order functions, especially in curried forms, can raise the level of programming dramatically. In addition, such languages often have straightforward operation semantics, thereby providing tremendous opportunities for parallel execution. Programs written in declarative languages thus eliminate the problem of "detecting parallelism". This paper illustrates programming in one such language, Id Nouveau, and contrasts it with programming in Fortran. Using an excerpt from an application known as Simple, it is shown how a program can be composed in Id Nouveau from small functions that directly relate to the mathematical and physical concepts of the problem. The difficulty of expressing these concepts in Fortran is discussed. Finally, it is shown that by performing simple transformations, such as in-line substitution of functions, the resulting Id Nouveau code becomes as efficient as an equivalent Fortran program written to run efficiently on a parallel machine.

Keywords: Dataflow, Id, Scientific Programming.

[Arvind and Nikhil, 1987]
Arvind and R. S. Nikhil.
Executing a Program on the MIT Tagged-Token Dataflow Architecture.
In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin, Also: Computation Structures Group Memo 271, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Abstract: The MIT Tagged-Token Dataflow project has an unconventional, but integrated approach to general-purpose high-performance parallel computing. Rather than extending conventional sequential languages, we use Id, a high-level language with fine-grained parallelism and determinacy implicit in its operational semantics. Id programs are compiled to dynamic dataflow graphs, a parallel machine language. Dataflow graphs are directly executed on the MIT Tagged-Token Dataflow Architecture (TTDA), a novel multiprocessor architecture. Dataflow research has advanced significantly in the last few years; in this paper we provide an overview of our current thinking, by describing example Id programs, their compilation to dataflow graphs, and their execution on the TTDA. Finally, we describe related work and the status of our project.

Keywords: Compilation, Dataflow, Id.

[Arvind *et al.*, 1986]
Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali.
I-Structures: Data Structures for Parallel Computing.
In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction, Proceedings of a Workshop*, volume 279 of *Lecture Notes in Computer Science*, pages 336–369, Santa Fe, New Mexico, September 29 – October 1, 1986. Springer, Berlin, Also: Computation Structures Group Memo 269, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Abstract: It is difficult simultaneously to achieve elegance, efficiency and parallelism in functional programs that manipulate large data structures. We demonstrate this through careful analysis of program examples using three common functional data-structuring approaches — lists using `Cons` and arrays using `Update` (both fine-grained operators), and arrays using `make-array` (a "bulk" operator). We then present I-structures as an alternative, defining precisely the parallel operational semantics of Id, a language with I-structures. We show elegant, efficient and parallel solutions for the program examples in Id. I-structures make the language non-functional, but do not raise determinacy issues. Finally, we show that even in the context of purely functional languages, I-structures are invaluable for implementing functional data abstractions.

Keywords: Dataflow, I-structures.

[Arvind *et al.*, 1988a]
Arvind, David E. Culler, and Gino K. Maa.
Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs.
Computation Structures Group Memo 279, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1988.

Abstract: A method for assessing the benefits of fine-grain parallelism in "real" programs is presented. The method is based on *parallelism profiles* and *speedup curves* derived by executing dataflow graphs on an interpreter under progressively more realistic assumptions about processor resources and communication costs. It is shown that programs, even using traditional algorithms, exhibit ample parallelism when parallelism is exposed at all levels, i.e., within expressions, across nested loops and function calls, and in producer-consumer relationships on individual elements of data structures. Since we only consider dataflow graphs compiled from the high-level language Id, the bias introduced by the language and the compiler is examined. A method of estimating speedup trough analysis of the ideal parallelism profile is developed, avoiding repeated execution of programs. It is shown that fine-grain parallelism can be used to mask large, unpredictably memory latency and synchronization waits in architectures employing dataflow instruction execution mechanisms. Finally, the effects of grouping portions of dataflow programs, such as function invocations or loop iterations, and requiring that the operators in a group execute on a single processor, are explored.

Keywords: Dataflow, Id, Parallelism Profiles.

[Arvind *et al.*, 1988b]
Arvind, Steve Heller, and Rishiyur S. Nikhil.
Programming Generality and Parallel Computers.
In *Fourth International Symposium on Biological and Artificial Intelligence Systems*, Trento, Italy, September 18–22, 1988. Also: Computation Structures Group Memo 287, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Abstract: In recent years, many "parallel, general-purpose" computers have become available, using various interconnections of processors and memories. However, extensions to conventional languages to program these machines have made programming significantly more complex

— a clear regression in software technology. The user now has the additional problem of partitioning his program into parallel parts, and often this is very machine-specific. Further, it is no longer easy to write determinate programs, making debugging a nightmare. In this paper we present an alternative approach. Id is a declarative, implicitly parallel language that simultaneously raises the level of programming and reveals much more parallelism than is possible with programmer annotations. Determinacy is guaranteed by the language semantics. We demonstrate this by developing a program in Id for a non-trivial problem called the "paraffins problem", and examining the available parallelism, after the fact.

Keywords: Dataflow, Id, I-Structures.

[Arya, 1989]
Kavi Arya.
Processes in a Functional Animation System.
In *FPCA '89, Functional Programming Languages and Computer Architecture*, pages 382–395, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: We investigate the problem of rapidly prototyping animation sequences. Conventional programming languages such as 'C' result in programs that are notoriously difficult to develop and to change. We show how a functional approach leads to a fresh perspective on the problem. A compact set of primitive operations over movies (i.e. *picture sequences*) is used as the basis for an animation system. We investigate the problem of interaction between components of an animation sequence and then show how lazy evaluation may be used to devise a process-oriented metaphor within a purely functional framework. This allows to arrange the interaction of animation sequences by encapsulating them within processes.

Keywords: Operating Systems, Process Networks.

[Ashcroft and Wadge, 1977]
E. A. Ashcroft and W. W. Wadge.
Lucid, a Nonprocedural Language with Iteration.
Communications of the ACM, 20(7):519–526, July 1977.

Abstract: Lucid is a formal system in which programs can be written and proofs of programs carried out. The proofs are particularly easy to follow and straight-forward to produce because the statements in a Lucid program are simply axioms from which the proof proceeds by (almost) conventional logic reasoning, with the help of a few axioms and rules of inference for the special lucid functions. As a programming language, Lucid is unconventional because, among other things, the order of statements is irrelevant and assignment statements are equations. Nevertheless, Lucid programs need not look much different than iterative programs in a conventional structured programming language using assignment and conditional statements and loops.

Keywords: Dataflow, Lucid, Formal Systems, Semantics, Proving.

[Ashcroft *et al.*, 1991]
E. A. Ashcroft, A. A. Faustini, and R. Jagannathan.
An Intensional Language for Parallel Application Programming.
In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 2, pages 11–50, ACM Press, New York, 1991.

Abstract: Programming languages that are not based on a sequential model of computation — i.e., declarative languages — can exhibit significant amount of implicit parallelism. The declarative languages are therefore important parallel programming languages. Within the declarative paradigm, we feel that the languages based on intensional logic are unique in that they permit data structures such as arrays, lists, and trees to be implemented in a manner that is easily distributable. Also, intensional programs are concise and elegant. To illustrate the concepts, we have chosen a particular intensional language, namely Lucid. Lucid's simplicity and semantic elegance make possible the inclusion of nontrivial application programs — an adaptive solution to the n-body problem and programs for LU decomposition of matrices. We present an analysis of the implicit parallelism available in these programs and show ways in which we can harness this parallelism.

Keywords: Dataflow, Lucid.

[Augustsson and Johnsson, 1989]
Lennart Augustsson and Thomas Johnsson.
Parallel Graph Reduction with the $\langle v, G\rangle$-machine.
In *FPCA '89, Functional Programming Languages and Computer Architecture*, pages 202–213, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: We have implemented a parallel graph reducer on a commercially available shared memory multiprocesor (a Sequent Symmetry) that achieves real speedup compared to a fast compiled implementation of the conventional G-machine. Using 15 procesors, this speedup ranges between 5 and 11 depending on the program. Underlying the implementation is an abstract machine called the $\langle v, G\rangle$-machine. We describe the sequential and the parallel $\langle v, G\rangle$-machine, and our implementation of them. We provide performance and speedup figures and graphs.

Keywords: Programmed Graph Reduction.

[Banatre and Le Metayer, 1989]
Jean-Pierre Banatre and Daniel Le Metayer.
Chemical Reaction as a Computational Model.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 103–117, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

Abstract: We present a new formalism called GAMMA which relies on the chemical reaction metaphor; the only data structure is the multiset and the computation can be seen as a succession of chemical reactions consuming elements of the multiset and producing new elements according to specific rules. We show the relevance of this formalism with respect to program development by proposing a systematic program derivation method.

Keywords: Multiset, Non-Determinism.

[Banatre and Le Metayer, 1990]
J.-P. Banatre and D. Le Metayer.
The GAMMA Model and its Discipline of Programming.
Science of Programming, 15(1):55–77, 1990
Keywords: Non-Determinism.

[Banatre *et al.*, 1988]
Jean-Pierre Banatre, Anne Coutant, and Daniel Le Metayer.
A Parallel Machine for Multiset Transformation and Its Programming Style.
Future Generation Computer Systems, 4:133–144, 1988
Keywords: Non-Determinism.

[Baraki, 1991]
Gebreselassie Baraki.
A Note on Abstract Interpretation of Polymorphic Functions.
In John Hughes, editor, *Functional Programming Languages and Computer Architectures*, volume 523 of *Lecture Notes in Computer Science*, pages 367–378, Harvard, Massachusetts, USA, 1991. Springer, Berlin.

Abstract: Strictness analysis of monomorphic functional programs by abstract interpretation has been extensively studied. Abramsky's work describes the extension to polymorphic functions. However, if different instances of a polymorphic function are used in a program, we have to re-analyze to compute abstract functions at each instance used. In the first-order case, Hughes showed how to compute the needed abstract functions without re-analysis. In this paper we develop a method of computing safe approximations to abstract functions of higher-order functions. The approximations are computed from abstract functions of simple instances of polymorphic functions. In the first-order case, the method gives exact values.

Keywords: Abstract Interpretation.

[Barendregt *et al.*, 1987a]
H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plassmeijer, and M. R. Sleep.
Term Graph Rewriting.

In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87 Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

Abstract: Graph rewriting (also called reduction) as defined by Wadsworth was introduced in order to be able to give a more efficient implementation of functional programming languages in the form of lambda calculus or term rewrite systems: identical subterms are shared using pointers. Several other authors, e.g. Ehrig, Staples, Raoult and van den Broek have given mathematical descriptions of graph rewriting, usually employing concepts from category theory. These papers prove among other things the correctness of graph rewriting in the form of the Church-Rosser property for "well-behaved" (i.e. regular) rewrite systems. However, only Staples has formally studied the soundness and completeness of graph rewriting with respect to term rewriting. In this paper we give a direct operational description of graph rewriting that avoids the category theoretic notions. We show that if a term t is interpreted as graph $g(t)$ and is reduced in the graph world, then the result represents an actual reduct of the original term t (*soundness*). For weakly regular term rewrite systems, there is also a *completeness* result: every normal form of a term t can be obtained from the graphical implementation. We also show completeness for all term rewrite systems which possess a so called hypernormalizing strategy, and in that case the strategy also gives a normalizing strategy for the graphical implementation. Besides having nice theoretical properties, weakly regular systems offer opportunities for parallelism, since redexes at different places can be executed independently or in parallel, without affecting the final result.

Keywords: Graph Reduction, Semantics.

[Barendregt *et al.*, 1987b]
H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plassmeijer, and M. R. Sleep.
Towards an Intermediate Language based on Graph Rewriting.
In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87 Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 159–175, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

Abstract: Lean is an experimental language for specifying computations in terms of graph rewriting. It is based on an alternative to Term Rewriting Systems (TRS) in which the terms are replaced by graphs. Such a Graph Rewriting System (GRS) consists of a set of graph rewrite rules which specify how a graph may be rewritten. Besides supporting functional programming, Lean also describes imperative constructs and allows the manipulation of cyclic graphs. Programs may exhibit non-determinism as well as parallelism. In particular, Lean can serve as an intermediate language between declarative languages and machine architectures, both sequential and parallel.

Keywords: Graph Rewriting, Semantics, Abstract Machine.

[Barth *et al.*, 1991]
Paul S. Barth, Rishiyur S. Nikhil, and Arvind.
M-Structures: Extending a Parallel, Non-strict, Functional Language with State.
In John Hughes, editor, *Functional Programming Languages and Computer Architectures*, volume 523 of *Lecture Notes in Computer Science*, pages 538–568, Harvard, Massachusetts, 1991. Springer, Berlin.

Abstract: Functional programs are widely appreciated for being "declarative" (algorithms are not *over-specified*) and implicitly parallel. However, when modeling state, both properties are compromised because the state variables must be "threaded" through most functions. Further, state "updates" may require much copying. A solution is to introduce assignments, as in ML and Scheme; however, for meaningful semantics, they resort to strict, sequential evaluation. We present an alternative approach called *M-structures*, which are imperative data structures with implicit synchronization. M-structures are added to Id, a parallel non-strict functional language. We argue that some programs with M-structures improve on their functional counterpart in three ways; they are (1) more *declarative*; (2) more *parallel*; and (3) more *storage efficient*. Points (1) and (2) contradict conventional wisdom. We study two problems: histogramming a tree, and graph traversal, comparing functional and M-structure solutions.

Keywords: State, Updatable arrays, Non-determinism.

[Bennett and Kelly, 1992]
Andrew J. Bennett and Paul H. J. Kelly.
Simulation of Multicache Parallel Graph Reduction.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: Parallel Graph Reduction is a simple formal model for parallel program execution which uses the shared-memory abstraction for all communication and synchronization between processors. Shared memory is used under a strict access regime with single assignment and blocking reads. In this paper we present the design and implementation of an efficient and accurate multiprocessor simulation scheme suitable to work with a parallel graph reducer and use the simulator to study the performance and pattern of access of a suite of benchmark programs. Threads are never migrated to another processor by the parallel graph reducer, and the effect of this scheduling policy is investigated

Keywords: Parallel Graph Reduction, Simulation.

[Berkling, 1975]
K. J. Berkling.
Reduction Languages for Reduction Machines.
In *2nd International Symposium on Computer Architecture*, pages 133–140. ACM/IEEE 75CH0916-7C, 1975
Keywords: Graph Reduction.

[Berrry *et al.*, 1992]
Dave Berrry, Robin Milner, and David Turner.
A Semantics for ML Concurrency Primitives.
In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 119–129. ACM Press, New York, 1992
Keywords: ML, Threads, Semantics.

[Berry and Boudol, 1990]
Gerard Berry and Gerard Boudol.
The Chemical Abstract Machine.
In *Seventeenth Annual ACM Symposium onn Principles of Programming Languages*, pages 81–94, San Francisco, California, January 17–19, 1990. ACM Press, New York.

Abstract: We introduce a new kind of abstract machine based on the chemical metaphor used in the Γ language of Banatre & al. States of a machine are chemical solutions where floating molecules can interact according to reaction rules. Solutions can be stratified by encapsulating subsolutions within membranes that force reactions to occur locally. We illustrate the use of this model by describing the operational semantics of the TCCS and CCS process calculi. We also show how to extract a higher-order concurrent λ-calculus out of the basic concepts of the chemical abstract machine.

Keywords: Abstract Machine.

[Bevan *et al.*, 1989]
D. I. Bevan, G. L. Burn, R. J. Karia, and J. D. Robson.
Principles For the Design of a Distributed Memory Architecture for Parallel Graph Reduction.
The Computer Journal, 32(5):461–469, 1989.

Abstract: Many models for the parallel reduction of lazy functional languages have been proposed in the literature. The one we have chosen to implement is based on evaluation transformers. An evaluation transformer says how much evaluation can be done to an argument expression in a function application, given the amount of evaluation that can be done to the application. Rather than just selecting a distributed memory architecture and trying to support parallel graph reduction, we investigate the implication of a minimally specified distributed memory architecture for parallel graph reduction. The results of the investigation are incorporated into an abstract machine which is able to support the communication and synchronisation needs of the parallel reduction model on a distributed memory architecture. Certain flags are needed on the nodes in the program graph in order to support the model. These are motivated and described.

Keywords: Parallel Graph Reduction, Evaluation Transformers.

[Bird and Wadler, 1988]
Richard Bird and Philip Wadler.
Introduction to Functional Programming.
Series in Computer Science, Prentice-Hall International, Englewood Cliffs, NJ, 1988
Keywords: Functional Programming, Streams.

[Bird, 1984]
R. S. Bird.
Using Circular Programs to Eliminate Multiple Traversals of Data.
Acta Informatica, 21:239–250, 1984.

Abstract: This paper describes a technique for transforming functional programs that repeatedly traverse a data structure into more efficient alternatives that do not. The transformation makes essential use of lazy evaluation and local recursion (such as provided by letrec, or its equivalent) to build a circular program that, on one pass over the structure, determines the effects of the individual traversals and the combines them.

Keywords: Circular Programs.

[Bjerner and Holmström, 1989]
Bron Bjerner and Sören Holmström.
A Compositional Approach to Time Analysis of First Order Functional Programs.
In *FPCA '89 Functional Programming Languages and Computer Architecture*, pages 157–165, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: We present a method for computing the number of steps needed to compute a lazy first order functional program `e` (to an approximation of its value). The method itself is described as a lazy functional program. Moreover, it is *compositional*, because it is defined by recursion on the structure of `e`. In other words, the number of steps needed to compute `e` is described in terms of the number of steps the parts of `e` need. (This is in contrast with the obvious operational method to define an interpreter and count the number of steps that it needs.) The equations that define the analyzing program can also be used when reasoning about time complexity of lazy functional programs. Two simple examples are given at the end of the paper.

Keywords: Time Analysis.

[Bjerner, 1989]
B. Bjerner.
Time Complexity of Programs in Type Theory.
PhD thesis, Chalmers University of Technology, Sweden, 1989
Keywords: Granularity Analysis.

[Blelloch *et al.*, 1993]
Guy E. Blelloch, Siddharta Chatterjee, C. Hardwick Jonathan, Jay Sipelstein, and Marco Zagha.
Implementation of a Portable Nested Data-Parallel Language.
In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP*, pages 102–112, San Diego, California, May 19–22, 1993. ACM Press.

Abstract: This paper gives an overview of the implementation of NESL, a portable nested data-parallel language. This language and its implementation are the first to fully support nested data structures as well as nested data-parallel function calls. These features allow the concise description of parallel algorithms on irregular data, such as sparse matrices and graphs. In addition, they maintain the advantages of data-parallel languages: a simple programming model and portability. The current NESL implementation is based on an intermediate language called VCODE and a library of vectore routines called CVL. It runs on the Connection Machine CM-2, the Cray Y-MP C90, and serial machines. We compare initial benchmark results of

NESL with those of machine-specific code on these machines for three algorithms: least-squares line-fitting, median finding, and a sparse-matrix vector product. These results show that NESL's performance is competitive with that of machine-specific codes for regular dense data, and is often superior for irregular data.

Keywords: Nested Data Parallelism.

[Blelloch, 1993]
Guy E. Blelloch.
Nesl: A Nested Data-Parallel Language (Version 2.6).
Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

Abstract: This report describes NESL, a strongly-typed, applicative, data-parallel language. NESL is intended to be used as a portable interrface for programming a variety of parallel and vector supercomputers, and as a basis for teaching parallel algorith.s Parallelism is supplied through a simple set of data-parallel constructs based on sequences (ordered sets), including a mechanism for applying any function over the elements of a sequence in parallel and a rich set of parallel functions that manipulate sequences. NESL fully supports nested sequences and nested parallelism — the ability to take a parallel function and apply it over multiple instances in parallel. Nested parallelism is important for implementing algorithms with complex and dynamically changing data structures, such as required in many graph and sparse matrix algorithms. NASL also provides a mechanism for calculating the asymptotic running time for a program on various parallel machine models, including the parallel random access machine (PRAM). This is useful for estimating running times of algorithms on actual machines and, when teaching algorithms, for supplying a close correspondence beteen the code and the theoretical complexity. This report defines NESL and describes several examples of algorithms coded in the language. The examples include algorithms for median finding, sorting, string searching, finding prime numbers, and finding a planar convex hull. NESL currently compiles to an intermediate language called VCODE, which runs on the CRAY Y-MP, Connection Machine CM-2, and Encore Multimax. For many algorithms, the current implementation gives performance close to optimized machine-specific code for these machines.

Keywords: Nested Data Parallelism.

[Bloss and Hudak, 1986]
Adrienne Bloss and Paul Hudak.
Variations on Strictness Analysis.
In *1986 ACM Symposium on Lisp and Functional Programming*, pages 132–142, Cambridge, Massachusetts, August 4–6, 1986. ACM Press, New York.

Abstract: This paper discusses the analysis of first-order functional programs in order to answer the following four questions: what definitely will be evaluated after some expression e is evaluated, what definitely won't be evaluated after e is evaluated, what definitely was evaluated before e was evaluated, what definitely wasn't evaluated before e was evaluated.

Keywords: Strictness Analysis.

[Bloss *et al.*, 1988]
Adrienne Bloss, P. Hudak, and J. Young.
An Optimizing Compiler for a Modern Functional Language.
The Computer Journal, 31(6):152–161, 1988.

Abstract: One of the factors hindering the use of functional languages has been their relatively poor performance in comparison to more traditional languages such as C and Pascal. During the last decade tremendous progress has been made in building implementations of functional languages but the approaches adopted have employed specialist hardware and/or compiler optimizations that have been developed specifically for functional languages. Building specialist hardware may be the best long-term solution but in the short run it is possible to increase the use and acceptance of functional languages by exploiting the performance of commercially available machines. The goal of this project described in this paper has been to design an optimizing compiler that produces fast code for functional languages on conventional sequential and parallel machines.

Keywords: Compilation, Strictness Analysis, Termination Analysis, Destructive Aggregate Updating.

[Bloss, 1987]
Adrienne Bloss.
Update Analysis and the Efficient Implementation of Functional Aggregates.
In Gilles Kahn, editor, *FPCA '87, Functional Programming Language and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 26–38, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

Abstract: In this paper we introduce a technique for efficiently implementing updatable aggregates in sequential first-order lazy functional languages. The technique is based on *path analysis*, a compile time analysis that yields information about *order of evaluation of expressions* and produces speedups over both the naive implementation and another optimization technique involving *trailers*.

Keywords: Aggregate Updates.

[Böhm and Sargeant, 1989]
A. P. W. Böhm and J. Sargeant.
Code Optimization for Tagged-Token Dataflow Machines.
IEEE Transactions on Computers, pages 4–14, January 1989
Keywords: Dataflow, Compilation, SISAL.

[Böhm *et al.*, 1991]
A. P. W. Böhm, D. C. Cann, J. T. Feo, and R. R. Oldehoeft.
SISAL 2.0 Reference Manual.
Technical Report CS-91-118, Computer Science Department, Colorado State University, Fort Collins, CO, November 1991
Keywords: SISAL, Single Assignment.

[Boyle *et al.*, 1986]
James M. Boyle, W. Dritz, Kenneth, M. N. Muralidharan, and Robert J. Taylor.
Deriving Sequential and Parallel Programs from Pure Lisp Specifications by Program Transformation.
In *Program Specification and Transformation*, pages 2–19, Bad Tölz, Germany, April 15–17, 1986. North Holland.

Abstract: We describe an "industrial strength" example of the use of specification and transformation to produce efficient programs. The specification is written in pure Lisp. Program transformations have been used to generate from it a program that executes efficiently on a wide variety of sequential machines. Another set of transformations has been used to generate programs from it that execute efficiently on global-memory parallel machines. We discuss some of the design descisions for the parallel version (and their motivations), optimization of parallel programs, and the benefits of the specification-transformation approach.

Keywords: Program Transformation, Parallelization, Lisp.

[Brandenburg *et al.*, 1988]
Joe Brandenburg, Mojy Mirashrafi, Jean McNamara, David Billstrom, Eric Gilbert, Mark Ferguson, Andrew Sizer, and Paul Alleyne.
Advances in Lisp for the iPSC/2.
Technical report, Intel Scientific Computers, Beaverton, Oregon, 1988.

Abstract: A complete redesign of Common Lisp for the Intel iPSC/2 sytem resulted in a new programming model based on multiple threads of execution, substantial performance gains exceeding all previously published timings for gabriel benchmarks, and a flexible user interface to manage the complexity of concurrently executing Lisp environments. Although multiple thread execution is not an unknown model of processing, the integration of threads with existing iPSC Lisp constructs provided new insight into certain concurrent applications. The performance of iPSC/2 Lisp is shown to be significantly faster than PSL on a Cray-XMP supercomputer, for two of the largest Gabriel benchmarks, iPSC/2 Lisp is shown to be the highest performance Lisp system available, commercially or in academia. The user interface allows direction and redirection of keyboard input, program output, and the handling of errors to multiple screens — important for large concurrent programs. Specific information is provided on iPSC/2 Lisp performance on the Gabriel Benchmarks; on the differences between iPSC/2 Lisp and CCLISP for the iPSC/2; and on the flexibility of the new user interface. Examples are given for each of the new constructs, user interface, and the performance.

Keywords: Lisp, Explicit Parallelism.

[Bratvold, 1992]
Tore A. Bratvold.
Determining Useful Parallelism in Higher Order Functions.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: Programs written in explicitly parallel languages are often composed of pieces of task specific code structured and interwoven with a limited number of well known parallel patterns. While such programs may indeed be very efficient in terms of execution speed, they are usually very hard both to write and reason about, and portability is low. The approach of expressing parallel patterns as templates or higher order functions has received attention lately, both from a theoretical point of view and in practical implementations. The paper presents an outline of the compilation stages in a proposed system based on this approach in combination with functional programming. The presence of useful parallelism is discussed for three higher order functions implemented as processor farms, and an example is presented to illustrate the approach.

Keywords: Higher Order Functions, Skeletons.

[Bratvold, 1993]
Tore A. Bratvold.
A Skeleton-Based Parallelising Compiler for ML.
In *5th International Workshop on the Implementation of Functional Languages*, pages 23–33, Numegen, The Netherlands, September, 1993.

Abstract: Languages with explicitly parallel constructs typically give the programmer detailed control of low level machine resources. Such control allows programs to be finely tuned to deliver high performances, but developing massively parallel programs from scratch in this way is a tedious and delicate task. Skeletons provide powerful abstractions over common patterns of parallel computation, and can be expressed as higher order functions (HOFs) in functional languages. In this paper we present SkelML, a skeleton-based compiler for ML, that exploits parallelism in certain HOFs and in function composition. Proviling information and performance models for skeletons are used to decide whether and how parallelism should be exploited. Issues of design and implementation are discussed, and an example is presented illustrating the compilation process and the performance of a prototype of the system.

Keywords: Algorithmic Skeletons, ML.

[Broy and Dendorfer, 1992]
Manfred Broy and Claus Dendorfer.
Modelling Operating System Structures by Timed Stream Processing Functions.
Journal of Functional Programming, 2(1):1–21, January 1992.

Abstract: Some extensions of the basic formalism of stream processing functions are useful to specify complex structures such as operating systems. In this paper we give the foundations of higher order stream processing functions. These are functions which send and accept not only messages representing atomic data, but also complex elements such as functions. Some special notations are introduced for the specification and manipulation of such functions. A representation of time is outlined, which enables us to model time dependent behavior. Finally, we demonstrate how characteristic operating system structures can be modelled by timed higher order stream processing functions.

Keywords: Operating System, Streams.

[Buelck and others, 1993]
T. Buelck et al.
Preliminary Experience with a π-RED Implementation on an nCUBE/2 System.

In *5th International Workshop on the Implementation of Functional Languages*, pages 101–113, Nijmegen, The Netherlands, 1993
Keywords: Graph Reduction, Distributed Memory.

[Burn and Le Metayer, 1992]
Geoffrey Burn and Daniel Le Metayer.
Cps-Translation and the Correctness of Optimising Compilers.
Technical Report DoC92/20, Department of Computing, Imperial College, London, UK, 1992.

Abstract: We show that compiler optimizations based on strictness analysis can be expressed formally in the functional framework using continuations. This formal presentation has two benefits: it allows us to give a rigorous correctness proof of the optimized compiler; and it exposes the various optimizations made possible by strictness analysis. These benefits are especially significant in the presence of partially evaluated data structures.

Keywords: Strictness Analysis, Optimization, Correctness Proof.

[Burn *et al.*, 1986]
Geoffrey L. Burn, Chris Hankin, and Samson Abramsky.
Strictness Analysis for Higher-Order Functions.
Science of Computer Programming, 7:249–278, 1986, Also: Graph Reduction, Proceedings of a Workshop, volume 217 of Lecture Notes of Computer Science, pages 42–63, Springer, Berlin, 1985.

Abstract: Abstract Interpretation is a compile-time technique which is used to gain information about a program that may then be used to optimize the execution of the program. A particular use of abstract interpretation is the strictness analysis of functional programs. This provides the key to the exploitation of parallelism in the evaluation of programs written in functional languages. In a language that has lazy semantics, the main potential for parallelism arises in the evaluation of operands of strict operators. A function is strict in an argument if its value is undefined whenever the argument is undefined. If we can use strictness analysis to detect which arguments a function is strict in, we then know that these arguments can be safely evaluated in parallel because this will not affect the lazy semantics. Experimental results suggest that this leads to significant speedups. Mycroft was the first person to apply abstract interpretation to the strictness analysis of functional programs. His framework only applies to first-order functions on flat domains. Many workers have proposed practical approaches to strictness analysis of higher-order functions over flat base domains but their work has not been accompanied by extensions to Mycroft's theoretical framework. In this paper, we give sound mathematical foundations for this work and discuss some of the practical issues involved. The practical approach is proved correct in relation to the theoretical framework.

Keywords: Strictness Analysis, Abstract Interpretation.

[Burn, 1987a]
G. L. Burn.
Abstract Interpretation and the Parallel Evaluation of Functional Languages.
PhD thesis, Department of Computing, Imperial College, London, 1987.

Abstract: A semantic criterion is presented which restricts the expressions which are evaluated to those which would eventually have been evaluated using a lazy evaluator. The complete analysis may be implemented as one of the passes in a compiler, so that programs can be automatically annotated with parallelism information.

Keywords: Abstract Interpretation, Evaluation Transformers.

[Burn, 1987b]
Geoffrey L. Burn.
Evaluation Transformers — A Model for the Parallel Evaluation of Functional Languages (Extended Abstract).
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 446–470, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

Abstract: If we are not careful, a parallel machine may become swamped with the computation of the expressions whose values are not needed in order to produce the result from a program. We give a semantic criterion which ensures that this does not happen. An abstract interpretation can be developed which gives the definedness of a function in terms of the definedness of its arguments. Traditionally this has been used to give a *strictness analysis* which has been interpreted to say how much the argument in an application can be evaluated in parallel with the application while still satisfying the semantic criterion. Strictness analysis however only takes into account local information. By taking into account contextual information of an application, we are able to find that in many cases more evaluation of the argument is allowed. This information is available from the same abstract interpretation that is used for strictness analysis. Given how much evaluation is allowed of a function application, an *evaluation transformer* says how much evaluation of the argument in the application is allowed. This leads to a natural model of the parallel evaluation of functional languages.

Keywords: Strictness Analysis, Abstract Interpretation, Evaluation Transformers.

[Burn, 1988]
G. L. Burn.
A Shared Memory Parallel G-machine Based on the Evaluation Transformer Model of Computation.
Draft, GEC Hirst Research Centre, Wembley, UK, November 1988.

Abstract: The evaluation transformer model of parallel reduction is a natural extension of lazy evaluation. It is therefore sensible to use the G-machine as a basis for defining a parallel machine supporting the model. Information from the model is compiled into the code for the parallel G-machine, and uses a number of special instructions defined for creation and sysnchronization of subtasks. The instructions from the sequential machine remain essentially unchanged. Besides the evaluation transformer model, we incorporate a number of important concepts into our definition that are not present in other definitions of parallel G-machines. A full definition of the compilation schemes and transition rules for the instructions are given in an appendix.

Keywords: Parallel Graph Reduction, Evaluation Transformers, G-Machine.

[Burn, 1990a]
G. L. Burn.
Implementing Lazy Functional Languages on Parallel Architectures.
In P. C. Treleaven, editor, *Parallel Computers — Object-Oriented, Functional, Logic*, Series in Parallel Computing, chapter 5, pages 101–140, John Wiley & Sons, Chichester, 1990.

Abstract: Lazy functional languages are a very powerful programming paradigm. In this chapter, we show how such languages, with no parallel constructs, can be implemented on parallel architectures. Troughout the project, we have taken a "language-first" approach to architecture development, allowing the programming language style and its semantics to drive the parallel implementation model and architectural development.

Keywords: Parallel Graph Reduction, Evaluation Transformers.

[Burn, 1990b]
Geoffrey Burn.
A New Domain For Head-Strictness.
Technical report, Department of Computing, Imperial College, London, UK, January 1990.

Abstract: A new domain is introduced that captures the notion of head-strictness in functional programming languages.

Keywords: Strictness Analysis, Abstract Interpretation.

[Burn, 1990c]
Geoffrey L. Burn.
A Relationship Between Abstract Interpretation and Projection Analysis (Extended Abstract).
In *POPL Symposium on Principles of Programming Languages, San Francisco, CA, January 17–19*, 1990.

Abstract: Abstract interpretation and projection analysis are two techniques for finding out information about lazy functional programs. Two typical uses of these techniques are speeding up sequential implementations, and the introduction of parallelism into parallel implementations. Our main result is the proof of a relationship between a certain class of projections and a certain class of abstract interpretations. One of the claims of projection analysis is that it can find out information about head-strictness, whilst abstract interpretation cannot. We show that there are at least two intuitive notions of head-strictness, and that one of them can be determined using abstract interpretation.

Keywords: Strictness Analysis, Abstract Interpretation, Projection Analysis.

[Burn, 1990d]
Geoffrey L. Burn.
Using Projection Analysis in Compiling Lazy Functional Programs.
In *ACM Conference on Lisp and Functional Programming*, pages 227–241, Nice France, June 27–29, 1990. ACM Press, New York.

Abstract: Projection Analysis is a technique for finding out information about lazy functional programs. We show how the information obtained from this analysis can be used to speed up sequential implementations, and introduce parallelism into parallel implementations. The underlying evaluation model is evaluation transformers, where the amount of evaluation that is allowed of an argument in a function application dependsss on the amount of evaluation allowed to the application. We prove that the transformed programs preserve the semantics of the original programs. Compilation rules, which encode the information from the analysis, are given for sequential and parallel machines.

Keywords: Strictness Analysis, Projection Analysis, Evaluation Transformers.

[Burn, 1991a]
Geoffrey Burn.
Lazy Functional Languages: Abstract Interpretation and Compilation.
Research Monographs in Parallel and Distributed Computing, Pitman, London, 1991.

Abstract: The author's Ph.D. thesis describes in large detail the concepts of abstract interpretation for the semantic analysis of functional languages. This book introduces the evaluation transformers approach to strictness analysis.

Keywords: Abstract Interpretation, Evaluation Transformers.

[Burn, 1991b]
Geoffrey L. Burn.
Implementing the Evaluation Transformer Model of Reduction on Parallel Machines.
Journal of Functional Programming, 1(2), April 1991, Also Technical Report DOC 90/24, Department of Computing, Imperial College, London, UK.

Abstract: The evaluation transformer model of reduction generalises lazy evaluation in two ways: it can start the evaluation of expressions before their first use, and it can evaluate expressions further than weak head normal form. Moreover, the amount of evaluation required of an argument to a function may depend on the amount of evaluation required of the function application. It is a suitable candidate model for implementing lazy functional languages on parallel machines. In this paper we explore the implementation of lazy functional languages on parallel machines, both shared and distributed memory architectures, using the evaluation transformer model of reduction. We will see that the same code can be produced for both styles of architecture, and the definition of the instruction set is virtually the same for each style. The essential difference is that a distributed memory architecture has one extra node type for non-local pointers and instructions which involve the value of such nodes need their definitions extended to cover this new type of node. To make our presentation accessible, we base our description on a variant of the well-known G-machine, an abtract machine for executing lazy functional programs.

Keywords: Strictness Analysis, Evaluation Transformers.

[Burn, 1991c]
Geoffrey L. Burn.
The Evaluation Transformer Model of Reduction and Its Correctness.
In *TAPSOFT 91, Brighton, UK, April 9-12*, 1991, Also: Technical Report DOC 90/19, Department of Computing, Imperial College, London.

Abstract: Lazy evaluation of functional programs incurs time and memory overheads, and restricts parallelism, when compared with programs that are evaluated strictly. A number of analysis techniques, such as abstract interpretation and projection analysis, have been developed to find out information that can be used to alleviate these overheads. This paper formalises an evaluation model, the evaluation transformer model of reduction, which can use information form these analysis techniques, and proves that the resulting reduction strategies produce the same answers as those obtained using lazy evaluation.

Keywords: Strictness Analysis, Evaluation Transformers.

[Burton, 1984]
F. Warren Burton.
Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs.
ACM Transactions on Programming Languages and Systems, 6(2):159–174, April 1984.

Abstract: When evaluating a functional program on a network of processors, it is necessary to decide when parallelism is desirable, which work may be transferred to another processor, and in what form the work is to be transferred. If the wrong decisions are made, a parallel evaluation may require asymptotically more time than a sequential evaluation, owing to communication costs. The introduction of annotations to give the programmer control over the above decisions is proposed. The annotations and their effects are definied in terms of the lambda calculus. Each application must have one of three annotations. No other annotations are required. Examples and possible extensions to this work, including annotated combinators, are briefly considered.

Keywords: Para-Functional Programming, Annotations.

[Burton, 1985a]
F. W. Burton.
Controlling Speculative Computation in a Parallel Functional Programming Language.
In *Fifth International Conference on Distributed Computing Systems*, Denver, Colorado, 1985
Keywords: Speculative Parallelism.

[Burton, 1985b]
F. Warren Burton.
Speculative Computation, Parallelism and Functional Programming.
IEEE Computer, 34(12):1190–1193, December 1985
Keywords: Parallel Functional Programming.

[Burton, 1987]
F. W. Burton.
Functional Programming for Concurrent and Distributed Computing.
The Computer Journal, 30(5):437–450, 1987.

Abstract: There are at least two approaches to the desing of languages for parallel computing. One approach is to use functional or relational languages which are easy to read, write, transform and verify. The more conventional approach is to use procedural langauges which give a programmer a high degree of control over the run-time behavior of a program. There is a need to reconcile these two approaches in a language which permits both simplicity and efficiency. We propose a small and simple set of annotations (or pragmas) to control the run-time behavior of a functional progrma. The annotations allow a programmer to use three forms of parameter passing. The parameter-passing mechanisms correspond to passing by name, value and need in a sequential language. In addition, in a distributed system a programmer can specify that work should be done on the current processor, an arbitrary processor, or a particular processor such as the one containing a specific data item. The annotatins cannot affect the meaning (result) of a functional program, except for causing non-termination in some cases (which we view as an extreme form of inefficiency). This separation of meaning from control allows a program to be both simple and efficient. Since non-determinism appears to be unavoidable without significant loss of efficiency in a concurrent system, the interaction of the proposed annotations with non-determinism is briefly considered. The run-time behavior of an annotated functional program is similar to that of a procedural programs using message passing, semaphores ore rendezvous to control communication and synchronization.

Keywords: Para-functional programming, Annotations, Process Mapping, Non-determinism.

[Burton, 1988]
F. W. Burton.
Nondeterminism with Referential Transparency in Functional Programming Languages.
The Computer Journal, 31(3):243–247, 1988.

Abstract: Procedural programmers have found nondeterminism to be unavoidable in many parallel programs (e.g. operating systems). If functional programming languages are to be usable as general-purpose parallel-programming languages then provision of some nondeterministic language constructs appears to be necessary. Existing constructs for nondeterminism are not functional and are not compatible with the mathematical foundations of functional programming, which require that the value of a function be uniquely determined by the values of its arguments. We propose to solve this problem by placing the nondeterminism in pseudo-data. A program is passed an infinite tree of two-valued decisions, along with its input. These decisions may be fixed at run time, thereby permitting nondeterminism. Once fixed, a decision remains unchanged, so equivalent expressions must always have the same value. The approach generalizes so that a program can make use of other run-time information such as the current time or the current amounts of available storage.

Keywords: Non-Determinism, Referential Transparency.

[Burton, 1989]
F.W̃arren Burton.
Indeterminate Behavior with Determinate Semantics in Parallel Programs.
In *FPCA '89 Functional Programming Languages and Computer Architecture*, pages 340–346, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: A parallel program may be indeterminate so that it can adapt its behavior to the number of processors available, or at least so that low level timing issues are removed from the program. Indeterminate programs are hard to write, understand, modify or verify. They are impossible to debug, since they may not behave the same from one run to the next. We propose a new construct, a polymorphic abstract data type called an *improving value*, with operations that have indeterminate behavior but simple determinate semantics. These operations allow the type of indeterminate behavior required by many parallel algorithms. We define improving values in the context of a functional programming language, but the technique can be used in procedural languages as well.

Keywords: Non-determinism.

[Burton, 1991]
Warren F. Burton.
Encapsulating Non-Determinacy in an Abstract Data Type with Determinate Semantics.
Journal of Functional Programming, 1(1):3–20, January 1991.

Abstract: A parallel program may be indeterminate so that it can adapt its behavior to the number of processors available. Indeterminate programs are hard to write, understand, modify or verify. They are impossible to debug, since they may not behave the same from one run to the next. We propose a new construct, a polymorphic abstract data type called an *improving value*, with operations that have indeterminate behavior but simple determinate semantics. These operations allow the type of indeterminate behavior required by many parallel algorithms. We define improving values in the context of a functional programming language, but the technique can be used in procedural languages as well.

Keywords: Non-determinism.

[Bush and Gurd, 1985]
V. J. Bush and J. R. Gurd.
Transforming Recursive Programs for Execution on Parallel Machines.
In Jean-Pierre Jouannaud, editor, *FPCA '85, Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 350–367, Nancy, France, September 16–19, 1985. Springer, Berlin.

Abstract: A linear recursive scheme makes just one recursive call; when it is executed, the resulting computation is sequential. Under certain conditions, there is an equivalent double recursive (bilinear) scheme which contains two independent recursive calls. The computation corresponding to this is parallel; the recursion causes a tree-like expansion of function calls. This recursive form is useful for exploiting the large amounts of parallelism available in dataflow and reduction machine architectures. On the other hand, the number of processors in such a parallel machine may be limited, in which case it might be desirable to convert a bilinear scheme to linear form by reversing the direction of the above transformation. In fact, the optimum strategy may be to transform dynamically between the two recursive forms, depending on the run-time availability of parallel processing resources. This paper describes the conditions under which linear and bilinear recursive forms are equivalent and demonstrates the importance and utility of establishing such a correspondence.

Keywords: FP, Recursion, Parallelization.

[Busvine, 1993]
David Busvine.
Detecing Parallel Structures in Functional Programs.
PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, October 1993
Keywords: Algorithmic Skeletons.

[Cann, 1989]
D. C. Cann.
Compilation Techniques for High Performance Applicative Computation.
PhD thesis, Colorado State University, Computer Science Department, Fort Collins, CO, 1989
Keywords: SISAL, Compilation.

[Cann, 1991]
D. C. Cann.
Vectorization of an Applicative Language: Current Results and Future Directions.
In *Compcon 91*, pages 396–402, February 1991
Keywords: Compilation, SISAL.

[Cann, 1992]
D. C. Cann.
Retire Fortran? A Debate Rekindled.
Communications of the ACM, 35(8):81–89, August 1992.

Abstract: We compare the execution performance of Sisal, a functional language for large-scale scientific computing, and Fortran on a Cray Y-MP/864. We examine the functional programming paradigm, discuss its attribute and advantages, and highlight the alient features of Sisal. In the remaining sections we illustrate the potential inefficiencies of functional computing, present our most recent performance data, and give some closing remarks regarding Fortran and the future of high-speed computing.

Keywords: SISAL, Single Assignment, Performance Measurements.

[Cheese, 1985]
A. B. Cheese.
The Applicability of SKI(BC) Combinators in a Parallel Rewrite Rule Environment.
Master's thesis, Department of Computer Science, University of Manchester, UK, 1985
Keywords: Combinators.

[Chen *et al.*, 1991]
Marina Chen, Young-il Choo, and Jingke Li.
Crystal: Theory and Pragmatics of Generating Efficient Parallel Code.
In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 7, pages 255–308, ACM Press, New York, 1991.

Abstract: The Crystal project has focused on making the task of programming massively parallel machines practical while not sacrificing the efficiency of the target code. We believe that these goals can be achieved by starting from a high-level problem specification, through a sequence of optimizations tuned for particular parallel machines, leading to the generation of efficient target code with explicit communications or synchronization. Our approach to automation is to design a compiler that classifies source programs according to the communication primitives and their cost on the target machine and that maps the data structures to distributed memory, and then generates parallel code with explicit communication commands. Regarding those classes of problems for which the default mapping strategies of the compiler are inadequate, we provide special language constructs for incorporating domain specific knowledge and directing the compiler in its mapping.

Keywords: Compilation, Crystal.

[Chen, 1985]
Marina C. Chen.
Crystal: A Synthesis Approach to Programming Parallel Machines.
In Michael T. Health, editor, *First Conference on Hypercube Multiprocessors, Knoxville, Tennessee, August 26–27*, pages 87–107, 1985.

Abstract: This paper addresses two software issues in parallel processing: Fiirst, how to program parallel machines effectively, and second, how to realize the high performance promised by the parallel hardware technology. An overview of the language Crystal and a programming methodology for a class of computational problems are presented. Crystal provides high-level abstractions for the behavior of an ensamble of parallel processors, and is devised for the purpose of managing effectively the complexity of programming large parallel systems. To address the issue of effective utilization of hardware technologies, a design methodology is developed for syntehsizing efficient systolic algorithms and architectures from problem definitions.

Keywords: Dataflow, Crystal.

[Chuang and Goldberg, 1991]
Tyng-Ruey Chuang and Benjamin Goldberg.
Backward Analysis for Higher-Order Functions Using Inverse Images.
Technical report, Department of Computer Science, New York University, NY, February 1991.

Abstract: We propose a method for performing backward analysis on higher-order functional programming languages based on computing inverse images of functions over abstract domains. The method can be viewed as abstract interpretation done backward. Given an abstract semantics which supports forward analysis, we can transform it into an abstract semantics which performs backward analysis. We show that if the original semantics is correct and computable, then the transformed version of the abstract semantics is also correct and computable. More specifically, given a forward abstract semantics of a higher-order functional language which is expressed in terms of Scott-closed powerdomains, we derive a backward abstraction semantics which is expressed in terms of Scott-open powerdomains. The derivation is shown to be correct and the relationships between forward analysis and backward analysiss is established. We apply this method to the classic strictness analysis in functional languages and obtain promising results. We show that the time complexity of inverse image based backward analysis is not much worse than the forward analysis from which it is derived. We then compare this work with previous works of Wadler, Hughes, and Burn, showing that some special restrictions and constructions in previous works have natural interpretation in the Scott-closed/Scott-open powerdomain framework. A brief outline of applying the inverse image method to other backward semantics analysis is also given.

Keywords: Strictness Analysis, Abstract Interpretation, Backwards Analysis.

[Clack and Peyton Jones, 1985]
Chris Clack and Simon L. Peyton Jones.
Strictness Analysis — A Practical Approach.
In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 35–49, Nancy, France, September 16–19, 1985. Springer, Berlin.

Abstract: Significant improvements in performance arise if we can arrange for parallel execution of programs. The absence of side effects in functional languages allows concurrent evaluation of the program, but in lazy implementations this risks wasting work by evaluating expressions which are subsequently discarded. We discuss the use of strictness analysis to determine at complie time which parts of program evaluation can safely be carried out concurrently. We give a practical explanation of this technique, concentrating particularly on the problem of finding fix points.

Keywords: Strictness Analysis, Abstract Interpretation, Fixed Points.

[Clarke, 1990]
Thomas Clarke.
The Semantics and Implementation of Aggregates or How to Express Concurrency without Destroying Determinism.
Technical report, Computer Laboratory, University of Cambridge, December 1990.

Abstract: Associative commutative operations lead to algorithms with concurrency which cannot be adequately represented by the pure lambda calculus. This paper investigates two classes of associative commutative operation which express extra concurrency, and shows that they preserve a simple denotational semantics. *Simple aggregates* encapsulate the evaluation order indeterminancy implicit in strict associative commutative operations and have no effect on semantics. *Early Readable Aggregates* encapsulate program values which are defined by a lower bound their denotations. The denotational join on ERA values is an associative commutative operation which can be used to express concurrency. Specific aggregates can be used to implement concurrent graph marking, time deterministic merge of lazy lists, and write-once locations.

Keywords: Non-determinism, Aggregate.

[Cohen *et al.*, 1983]
Shismon Cohen, Roni Rosner, and Ari Zidon.
PARALISP Simulator (Reference Manual).
Research Report 83-2, Computer Science Department, Hebrew University, Jerusalem, Israel, January 1983.

Abstract: PLISP ("Parallel Lisp") is the current implementation of "Senile Parallel Execution of Lisp Programs" (Lehmann and Cohen). PLISP may be thought of as a virtual machine designed to evaluate LISP expressions in a parallel fashion. The language of this virtual machine is PARALISP. This means, for example, that when a function application contains several formal arguments, these arguments may be evaluated in parallel, under the assumption that there are no dependencies, or side effects. Therefore, PARALISP is a version of LISP designed to enable parallel evaluation to occur without side effects.

Keywords: Lisp.

[Cole, 1988]
M. I. Cole.
A 'Skeletal' Approach to the Exploitation of Parallel Computation.
In *CONPAR 88*, volume C, pages 100–107, Manchester, UK, September 12–16, 1988. British Computer Society.

Abstract: This paper introduces a new approach to the design of programming ystems which can exploit parallelism while maintaining a reasonable level of abstraction from the underlying hardware. This is based on the concept of "algorithmic skeletons". The abstract specification of four such skeletons are presented and a possible implementation on a grid of processing units is outlined.

Keywords: Skeletons, Para-Functional Programming.

[Cole, 1989]
Murray I. Cole.
Algorithmic Skeletons: Structured Management of Parallel Computation.
Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989
Keywords: Parallel Functional Programming.

[Cooper and Morrisett, 1990]
Eric C. Cooper and J. Gregory Morrisett.
Adding Threads to Standard ML.
Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1990.

Abstract: We have added multiple threads of control to the Standard ML programming language. Stanford ML's support for first-class functions and automatic storage management influenced the design in a number of ways. We demonstrate how other concurrency and synchronization operations, such as cobegin/coend, futures, and events, can be implemented in terms of the thread interface. Finally, we describe three implementations of the thread interface: a coroutine version, a uniprocessor preemptive version, and a multiprocessor Mach-based version.

Keywords: ML, Threads.

[Cousot and Cousot, 1977]
Patrick Cousot and Radhia Cousot.
Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.
In *Fourth ACM Symposium on Principles of Programming Language*, pages 238–252, Los Angeles, California, January 1977. ACM Press, New York
Keywords: Abstract Interpretation.

[Cox *et al.*, 1989]
Stuart Cox, Hugh Glaser, and Mike Reeve.
Implementing Functional Languages on the Transputer.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 287–295, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

Abstract: The aim of work presented here is to develop a method for implementing lazy, curried functional languages for such architectures, giving performance comparable with imperative languages and supporting interlanguage working. The system described in this paper confirms to the interlanguage standard of the transputer and, for example, as a result it can use a functional language to map a C function over a list that has been constructed by the functional program.

Keywords: Parallel Graph Reduction.

[Culler *et al.*, 1991]
David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek.
Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine.
In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 26(4) of *SIGPLAN Notices*, pages 164–175, Santa Clara, California, April 8–11, 1991.

Abstract: In this paper, we present a relatively primitive execution model for fine-grain parallelism, in which all synchronization, scheduling, and storage management is explicit and under compiler control. This is defined by a threaded abstract machine (TAM) with a multilevel scheduling hierarchy. Considerable temporal locality of logically related threads is demonstrated, providing an avenue for effective register use under quasi-dynamic scheduling. A prototype TAM instruction set, TL0, has been developed along with a translator to a variety of existing sequential and parallel machines. Compilation of Id, an extended functional language requiring fine-grain synchronization, under this model yields performance approaching that of conventional languages on current uniprocessors. Measurements suggest that the net cost of synchronization on conventional multiprocessors can be reduced to within a small factor of that on machines with elaborate hardware support, such as proposed dataflow architectures. This brings into question whether tolerance to latency and inexpensive synchronization require specific hardware support or merely an appropriate compilation strategy and program representation.

Keywords: Multi-Threaded Machines, Dataflow, Compilation.

[Cupitt, 1988]
J. Cupitt.
Another New Scheme for Writing Functional Operating Systems.
Technical Report 52, Cumputing Laboratory, University of Kent, Canterbury, UK, 1988
Keywords: Operating System.

[Cupitt, 1989a]
J. Cupitt.
A Brief Walk Through KAOS.

Technical Report 58, Computing Laboratory, University of Kent, Canterbury, UK, 1989
Keywords: Operating System.

[Cupitt, 1989b]
J. Cupitt.
The Design and Implementation of an Operating System in a Functional Language.
PhD thesis, University of Kent, October 1989
Keywords: Operating System.

[Darlington and Reeve, 1981]
John Darlington and Mike Reeve.
ALICE — A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages.
In *FPCA '81, Functional Programming Languages and Computer Architecture*, pages 65–74, Boston, Massachusetts, 1981. ACM Press, New York.

Abstract: In this paper we present a scheme for the parallel evaluation of a wide variety of applicative languages and provide an overview of the architecture of a machine on which it may be implemented.

Keywords: Parallel Graph Reduction.

[Darlington *et al.*, 1988]
John Darlington, Hessam Kohshnevisan, Lee McLoughlin, Nigel Perry, Hellen Pull, Keith Sephton, and Lyndon While.
An Introduction to the Flagship Programming Environment.
In *CONPAR 88*, volume A, pages 75–86, Manchester, UK, September 12–16, 1988. British Computer Society.

Abstract: The Flagship programming environment provides support for the development, implementation and modification of programs. Its novelty is that it is designed to support programming in a pure functional language and to give mechanical assistance to the process of deriving programs from high level specifications via correctness preserving transformations. The target for these programs is both the Flagship Parallel Machine and conventional parallel machines.

Keywords: Program Transformation.

[Darlington *et al.*, 1989]
J. Darlington, M. Reeve, and S. Wright.
Programming Parallel Computer Systems using Functional Languages and Program Transformation.
In *Parallel Processing '89*, Leiden, The Netherlands, 1989.

Abstract: One of the major problems associated with parallel computer systems is the economical production of software that fully exploits the hardware. Functional languages facilitate a formal methodology for the development of programs for parallel machines. This paper provides an overview of one program development methodology developed at imperial College, and presents a case study that illustrates its operation and quantifies its benefits using results of experiments carried out on the ALICE parallel machine.

Keywords: Parrallel Graph Reduction.

[Darlington *et al.*, 1991]
J. Darlington, A. J. Field, P. G. Harrison, D. Harper, G. K. Jouret, P. J . Kelly, K. M. Sephton, and D. W. Sharp.
Structured Parallel Functional Programming.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 31–52, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: We present a methodology, based on functional program transformation, aimed at facilitating the development of applications on a variety of diverse parallel machines. The novelty of the method lies in employing a fixed repertoire of parallel program forms, or *skeletons*, expressed as higher-order functions for which efficient implementations existon particular parallel machines. Program transformation techniques are then used to convert general functional programs into compositions of known skeletons. We give an overview of the problems of programming parallel machines, highlighting the problem in both the implicit and explicit approach, present a taxonomyof parallel machines, discuss the role and advantage of using skeletons, give definitions of particular skeletons, discuss semantic issues, detail the transformations employed and the implementation route used and give examples of the technique in action. The methodology is successful in allowing a broad variety of parallel machines, from SIMD to MIMD, to be programmed efficiently in a uniform manner that still allows portability across different machine types.

Keywords: Parallel Programming Skeletons, Program Transformation.

[Darlington *et al.*, 1993a]
J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu.
Parallel Programming Using Skeleton Functions.
In *PARLE 93: Parallel Languages and Architectures Europe*, Munich, Germany, June 14–18, 1993
Keywords: Algorithmic Skeletons.

[Darlington *et al.*, 1993b]
J. Darlington, M. Ghanen, and H. W. To.
Structured Parallel Programming.
In *Working Conference on Massively Parallel Programming Models: Suitability, Realization, and Performance*, Berlin, Germany, September 20–23, 1993.

Abstract: We propose the use of a range of algorithmic forms, known as skeletons, which abtract useful parallel computational structures from applications. Applications are then constructed as instatiations of these algorithmic forms. Our skeletons are expressed in a functional language as higher order functions that can be defined in the language.

Keywords: Algorithmic Skeletons.

[Davis and Wadler, 1989]
K. Davis and P. Wadler.
Backwards Strictness Analysis: Proved and Improved.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 12–30, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

Abstract: Given a syntax tree representing an expression, and some information regarding that expression, a *backwards analysis* will involve propagating the information (with appropriate transformation) towards the leaves of the tree, to yield information about the subexpressions. Here, the information at the root will describe the required definedness of the value of the expression, with the results of the analysis describing the definedness of the values lower in the tree sufficient or necessary to meet the condition at the root. In *Projections for Strictness Analysis*, such an analysis is described in which the information at each node is encoded by a special kind of function called a *projection*, with the results of the analysis leaving strictness information about the expression. This paper describes a more general and powerful technique, and provides proofs that both techniques meet a corresponding generalization of the safety condition.

Keywords: Strictness Analysis, Backwards Analysis, Projections.

[Davis and Wadler, 1990]
K. Davis and P. Wadler.
Strictness Analysis in Four Dimensions.
In *Functional Programming: Proceedings of the 1990 Workshop*, Springer Workshops in Computing, Ullapool, Scotland, August 1990. Springer, Berlin.

Abstract: A relational, higher-order monomorphic backward method for strictness analysis is given.

Keywords: Strictness Analysis, Backward Analysis.

[Davis, 1991]
K. Davis.

A Note on the Choice of Domains for Projection-Based Program Analysis.
In Philip Wadler, editor, *Functional Programming: Proceedings of the 1991 Glasgow Workshop*, Springer Workshops in Computing, Isle of Skye, Scotland, August 13-15, 1991. Springer, Berlin
Keywords: Strictness Analysis, Projections.

[Debray *et al.*, 1990]
Saumya K. Debray, Nai-Wei Lin, and Manuel Hermenegildo.
Task Granularity Analysis in Logic Programs.
In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25(6) of *SIGPLAN NOTICES*, pages 174–188, White Plains, New York, June 20–22, 1990.

Abstract: While logic programming languages offer a great deal of scope for parallelism, there is usually some overhead associated with the execution of goals in parallel because of the work involved in task creation and scheduling. In practice, therefore, the "granularity" of a goal, i.e. an estimate of the work available under it, should be taken into account when deciding whether or not to executed a goal concurrently as a separate task. This paper describes a method for estimating the granularity of a goal at compile time. The runtime overhead associated with our approach is usually quite small, and the performance improvements resulting from the incorporation of grainsize control can be quite good. This is shown by means of experimental results.

Keywords: Granularity Analysis.

[Deschner, 1989]
J. M. Deschner.
Simulating Multiprocessor Architectures for Compiled Graph-Reduction.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 225–237, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

Abstract: Software simulations of hardware architectures are useful tools for evaluating a systems performance prior to its construction. This is especially true of multi-processor architectures where the design complexity is increased by orders of magnitude. It is also necessary to be able to simulate accurately the behavior of a parallel machine after it has been built. Statistics may then continue to be gathered without adversely affecting the behavior of the hardware. This paper describes in detail a system for simulating the parallel execution of functional programs based on compiled graph-reduction.

Keywords: Parallel Graph Reduction, Simulation.

[Dimitrovsky, 1988]
Isaac Dimitrovsky.
ZLISP 0.85 Reference Manual.
Ultracomputer Research Laboratory, NewYork, March 1988.

Abstract: ZLISP is a portable parallel LISP system for shared memory MIMD supercomputers. It contains several extensions to LISP to support the writing of parallel programs. In the current version, all parallelism is explicit although this may change in the future.

Keywords: Lisp, Futures.

[Dörr, 1992]
Heiko Dörr.
Monitoring with Graph-Grammars as Formal Operational Models.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: This paper presents a monitoring method which reduces the costs of event-tracing. To support the monitoring of a system-run, a model is used to manage the system's information which has been collected since the beginning of the run. By this means the collection of redundant event-trace information is avoided. The background and usage of model-based monitoring is discussed, too. Graph-grammars with grouped rules are proposed and defined as an operational formalism to model the behavior of a system. They are especially suitable for the description of distributed systems whose functionality can be characterized by structural changes. In graph-grammar models, relevant functional parts become directly addressable.

Therefore, system analysis becomes easier. The formal model of a parallel graph reduction machine is introduced in order to illustrate the monitoring method.

Keywords: Monitoring, Parallel Graph Reduction.

[Dybjer, 1987]
P. Dybjer.
Computing Inverse Images.
In *International Conference on Automata, Languages, and Programming*, 1987
Keywords: Strictness Analysis.

[Ekanadham and Arvind, 1987]
Kattamuri Ekanadham and Arvind.
SIMPLE: Part 1 — An Exercise in Future Scientific Programming.
Computation Structures Group Memo 273, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1987.

Abstract: Ideally, a high-level language should provide a way of writing abstractions which are as close to the problem domain as possible, as well as facilitate efficient implementations of these abstractions lest a user try to "get underneath" the abstractions. With the advent of parallel machines, a language such as FORTRAN fails on both counts. Functional and other declarative languages allegedly offer relief on both counts. The use of higher order functions, including the free use of curried forms, can dramatically raise the level of programming. In addition, such languages often have straight-forward operational semantics which admits tremendous opportunities for parallel execution. Programs in declarative languages, thus eliminate the problem of "detecting parallelism". In this report we have examined the first part of this claim by writing an application known as the SIMPLE code in a language called Id Nouveau. The issues of parallelism will be examined in a companion report, Part II, to be published later. We have presented a high-level description of the algorithms used in the hydrodynamics problem, called SIMPLE. Corresponding Id Noveau program fragments are derived from the mathematical descriptions. The emphasis is on the style of programming, and not on algorithmic cleverness. The resulting program has 550 lines and runs successfully in ID World, which is a compiler for Id Noveau together with an abstract simulation facility for a dataflow machine. This report presumes the knowledge of Id Noveau, functional programming and I-structures.

Keywords: Scientific Programming, Dataflow, Id, I-Structures.

[Ekanadham, 1991]
Kattamuri Ekanadham.
A Perspective on Id.
In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 6, pages 197–254, ACM Press, New York, 1991.

Abstract: This chapter is a comprehensive exposition of many aspects of the Id language. Section 6.2 introduces the principal features of the language to the reader unfamiliar with. Section 6.3 thus discusses three examples in detail. These are from numerical applications and are all self-contained. Section 6.4 gives the operational semantics for the language in terms of rewrite rules. Section 6.5 illustrates a few optimizations to show how they can be expressed in this framework. Section 6.6 discusses the lower level abstraction of I-structures which is used to implement the higher level abstractions.

Keywords: Dataflow, Id.

[Evripidou *et al.*, 1989]
P. Evripidou, W. Najjar, and J.-L. Gaudiot.
A Single-Assignment Language on a Distributed Memory Multiprocessor.
In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, volume 365 of *Lecture Notes in Computer Science*, pages 304–320, Eindhoven, The Netherlands, June 12–16, 1989. Springer, Berlin.

Abstract: Large-scale distributed memory multiprocessors have become commercially available and have proved to be a low-cost alternative to supercomputers for many scientific computations. The programming, and debugging, of such systems remains, however, a difficult and tedious task. Single-assignment and applicative languages have proved to be a serious alternative to imperative languages for the programming of parallel computer systems. They offer the advantage of a high level of programmability and eliminate the problem of detecting parallelism. Their functional property allows an asynchronous parallel execution that does not comprise the correctness of the computation. This paper describes the implementation of a single-assignment language, SISAL, on a distributed memory multiprocessor.

Keywords: Dataflow, SISAL, Compilation.

[Feeley and Miller, 1989]
Marc Feeley and James S. Miller.
A Parallel Virtual Machine for Efficient Scheme Compilation.
Technical report, Brandeis University, Waltham, Massachusetts, 1989.

Abstract: Programs compiled by Gambit, our Scheme compiler, achieve performance as much as twice that of the fastest available Scheme compilers. Gambit is easily ported, while retaining its high performance, through the use of a simple virtual machine (PVM). PVM allows a wide variety of machine-independent optimizations and it supports parallel computation based on the `future` construct. PVM conveys high-level information bidirectionally between the machine-independent front end of the compiler and the machine-dependent back end, making it easy to implement a number of common back end optimizations that are difficult to achieve for other virtual machines. PVM is similar to many real computer architectures and has an option to efficiently gather dynamic measurements of virtual machine usage. These measurements can be used in performance prediction for ports to other architectures as well as design decisions related to proposed optimizations and object representations.

Keywords: Lisp, Futures, Virtual Machine.

[Feo *et al.*, 1990]
J. T. Feo, D. C. Cann, and R. R. Oldehoeft.
A Report on the SISAL Language Project.
Journal of Parallel and Distributed Computing, 10(4):349–366, December 1990
Keywords: SISAL.

[Field and Harrison, 1988]
Anthony J. Field and Peter G. Harrison.
Functional Programming.
International Computer Science Series, Addison-Wesley, Wokingham, England, 1988.

Abstract: This book covers all aspects of functional programming languages including their implementation, compilation and optimization. It also discusses the functional description of process networks, streams, dataflow and strictness analysis by abstract interpretation.

Keywords: Functional Programming.

[Flajolet, 1987]
P. Flajolet.
Mathematical Tools for Automatic Program Analysis.
Technical Report 603, Institut National de Recherche en Informatique et en Automatique, February 1987.

Abstract: This paper examines generating functions to be used in a system for deriving expected complexities (average computation time).

Keywords: Granularity Analysis.

[Friedman and Wise, 1976]
D. P. Friedman and D. S. Wise.
The Impact of Applicative Multiprogramming on Multiprocessing.
In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 263–272, 1976
Keywords: Parallel Functional Programming.

[Friedman and Wise, 1978a]
D. P. Friedman and D. S. Wise.
Applicative Multiprogramming.

Technical Report 72, Computer Science Department, Indiana University, Bloomington, Indiana, December 1978
Keywords: Parallel Functional Programming.

[Friedman and Wise, 1978b]
Daniel P. Friedman and David S. Wise.
Aspects of Applicative Programming for Parallel Processing.
IEEE Computer, 27(4):289–296, April 1978.

Abstract: Early results of a project on compiling stylized recursion into stackless iterative code are reviewed as they apply to a target environment with multiprocessing. Parallelism is possible in executing the compiled image of argument evaluation (collateral argument evaluation in Algol 68), of data structure construction when suspensions are used, and of functional combinations. The last facility provides generally, concise expression for all operations performed in Lisp by mapping functions and in APL by typed operators; there are other uses as well.

Keywords: Combinators.

[Friedman and Wise, 1979]
D. P. Friedman and D. S. Wise.
A Constructor for Applicative Multiprogramming.
Technical Report 80, Computer Science Department, Indiana University, Bloomington, Indiana, January 1979
Keywords: Parallel Functional Programming.

[Friedman and Wise, 1980]
Daniel Friedman and David Wise.
An Indeterminate Constructor for Applicative Programming.
In *7th Symposium on Principles of Programming Languages*, 1980
Keywords: Non-Determinism.

[Gabriel and McCarthy, 1984]
Richard P. Gabriel and John McCarthy.
Queue-based Multi-processing Lisp.
In *1984 ACM Symposium on Lisp and Functional Programming*, pages 25–44, Austin, Texas, August 6–8, 1984. ACM Press, New York.

Abstract: A the need for high-speed computers increases, the need for multi-processors will become more apparent. One of the major stumbling blocks to the development of useful multi-processors has been the lack of a good multi-processing language — one which is both powerful and understandable to programmers. Among the most compute-intensive programs are artificial intelligence (AI) programs, and researchers hope that the potential degree of parallelism in AI programs is higher than in many other applications. In this paper we propose multi-processing extensions to Lisp. Unlike other proposed multi-processing Lisps, this one provides only a few very powerful and intuitive primitives rather than a number of parallel variants of familiar constructs.

Keywords: Lisp, Queues.

[Gabriel and McCarthy, 1988]
R. Gabriel and J. McCarthy.
Qlisp.
In J. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, pages 63–89, Kluwer Academic Publishers, 1988
Keywords: Lisp, Parallelization.

[Garsden and Wendelborn, 1990]
H. Garsden and A. L. Wendelborn.
A Comparison of Microtasking Implementation of the Applicative Langague SISAL.
In H. Burkhart, editor, *CONPAR 90 — VAPP IV, Joint International Conference on Vector and Parallel Processing*, volume 457 of *Lecture Notes in Computer Science*, pages 697–708, Zurich, Switzerland, September 10–13, 1990. Springer, Berlin.

Abstract: The applicative parallel programming language SISAL has been implemented on conventional multiprocessors with the use of microtasking systems designed to support fine-grained parallelism. Experiments with different versions of such a system, including an implementation under the Mach operating system using its lightweight processes, are reported.

Keywords: Sisal, Dataflow.

[Gaudiot and Lee, 1987]
J. L. Gaudiot and L. T. Lee.
Multiprocessor Systems Programming in a High-Level Data-Flow Language.
In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87, Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures*, volume 258 of *Lecture Notes in Computer Science*, pages 134–151, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

Abstract: The data-flow model of computation is an attractive methodology for multiprocessor programming for it offers the potential for unlimited parallelism detection at no programmer's expense. It is here applied to a distributed architecture based on a commercially available microprocessor (the Inmos Transputer). In this project, we have integrated the high-level data driven principles of scheduling within the Transputer architecture so as to provide high programmability of our multicomputer system. A completet programming environment which translates a complex data-flow program graph into occam has been developed and is presented in this paper. We here describe in detail the mapping from the SISAL high-level constructs into the low-level mechanisms of the Transputer (synchronization, structure representation, etc.). The partitioning issues (granularity of the graph) are presented and several solutions based upon both data-flow analysis (communication costs) and program syntax (program structure) are proposed and have been implemented in our programming environment. Finally, we present and analyze graph allocation and optimization schemes to improve the performance of the resulting occam program.

Keywords: SISAL, Dataflow.

[Gebreselassi and Hughes, 1989]
Baraki Gebreselassi and John Hughes.
Abstract Interpretation of Polymorphic Functions.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 31–40, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

Abstract: Abstract interpretation is one of the popular techniques used in doing program analysis. In this paper, we define an abstract interpretation of polymorphic functions which may be used to perform strictness analysis. The abstract functions of polymorphic functions are again polymorphic. Finally, it is proved that the abstract interpretation is safe.

Keywords: Abstract Interpretation, Strictness Analysis, Polymorphism.

[George, 1989]
Lal George.
An Abstract Machine for Parallel Graph Reduction.
In *FPCA '89, Functional Programming Languages and Computer Architecture*, pages 214–229, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: An abstract machine for parallel graph reduction on a shared memory multiprocessor is described. This is intended primarily for normal order (*lazy*) evaluation of functional programs. It is absolutely essential in such a design to adapt an efficient sequential model since during execution under limited resources available, performance will be reduced in the limit to that of the sequential engine. Parallel evaluation of normal order functional languages performed naively can result in poor overall performance despite the avilability of sufficient processing elements and parallelism in the application. Needless context switching, task migration and continuation building may occur when a sequential thread of control would have sufficed. Furthermore, the compiler using static information cannot be fully aware of the availability of resources and their optimal utilization at any moment in run time. Indeed this may vary between runs which further aggravates the job of the compiler writer in generating optimal and compact code for programs. The benefits derived from this model are: 1) it is based on the G-machine so that execution under limited resources will default to a performance close to that of the G-machine; 2) the additional instructions needed to control the complexities of paralllel evaluation are extremely simple, almost trivializing the job of the compiler writer; 3) attempts are made where possible to avoid context switching and task migration by retaining a sequential thread of control (made more clear in the paper), and 4) the method has demonstrated good overall performance on a shared memory multiprocessor.

Keywords: Parallel Graph Reduction, G-Machine.

[Giacalone *et al.*, 1989]
Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad.
Facile: A Symmetric Integration of Concurrent and Functional Programming.
International Journal of Parallel Programming, 18(2):121–160, April 1989
Keywords: Facile.

[Giorgi and Le Métayer, 1990]
J.-F. Giorgi and D. Le Métayer.
Continuation-Based Parallel Implementation of Functional Languages.
In *ACM Symposium on Lisp and Functional Programming*, Nice, France, June 25–29, 1990. ACM Press, New York
Keywords: Continuations.

[Glas *et al.*, 1992]
J. C. Glas, R. F. H. Hofman, and W. G. Vree.
Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: The referential transparency of functional languages imposes restraints on the communication between parallel tasks. This seriously hampers the implementation of parallel branch-and-bound algorithms, because the parallel tasks cannot update a shared bound asynchronously. Maintaining local bounds only causes enormous performance loss, because parallel branch-and-bound algorithms exploit speculative parallelism. Therefore, the parallel tasks have to exchange their bounds synchronously, but this lowers the average processor utilization. Three parallel functional implementations of a generally applicable branch-and-bound algorithm have been developed to investigate this trade-off. Experiments have been performed with the cargo loading problem. Two functional equivalents of imperative branch-and-bound approaches are outperformed by a dedicated parallel functional branch-and-bound implementation. A new speculative parallel annotation is proposed to tackle the trade-off between speculative transformation loss and average processor utilization on system level.

Keywords: Branch-and-Bound, Speculative Parallelism.

[Glauert *et al.*, 1988]
J. R. W. Glauert, K. Hammond, J. R. Kennaway, and G. A. Papadopoulos.
Using Dactl to Implement Declarative Languages.
In *CONPAR 88*, volume A, pages 87–94, Manchester, UK, September 12–16, 1988. British Computer Society.

Abstract: Dactl is a language of graph rewriting, intended as an intermediate language and compiler target language for highly parallel machines. It is based on a form of graph rewriting which can be used to implement functional, logic, and imperative languages.

Keywords: Intermediate Language, Compilation.

[Glauert, 1992]
John Glauert.
Parallel Implementation of Functional Languages Using Small Processes.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: We report work in progress on the implementation of languages that integrate concurrent and functional programming styles. A translation scheme is presented for mapping such languages into process networks using a simple notation based on communication between named processes. The translation of functional code is sequential, but parallelism arises from the use of concurrency constructs in the source language. Early implementation experiments have used graph rewriting techniques, although work on a more direct implementation is in progress.

Keywords: Process Networks, Para-Functional Programming, Parallel Graph Reduction.

[Glück and Demuth, 1987]
Robert Glück and Christian Demuth.
OC-FP: An Applicative Language Combination with Occam and the Algebra of Processes.
Microprocessing and Microprogramming, 21:549–558, 1987.

Abstract: A functional language based on the FP-model is presented. The language combines occam processes and is used as a command level language for occam systems. It is well suited as a graphical language for concurrent programming. An algebra, based on the functional properties of the language, makes process and communication structures accessible to (semi-)automatical transformations.

Keywords: FP, Occam, Process Algebra.

[Goldberg and Hudak, 1987]
Benjamin Goldberg and Paul Hudak.
Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor.
In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction, Proceedings of a Workshop*, volume 279 of *Lecture Notes in Computer Science*, pages 94–113, Santa Fe, New Mexico, September 29 – October 1, 1987. Springer, Berlin.

Abstract: Alfala is an implementation of a functional language on the Intel iPSC multiprocessor. It is based on a heterogeneous abstract machine model consisting of both graph reduction and stack oriented execution. Alfala consists of two major components, a compiler and a runtime system. The source language, Alfl, contains no constructs that allow the programmer to specify parallelism or synchronization and thus it is the task of the compiler to detect the exploitable parallelism in a program. The runtime system supports dynamic scheduling, interprocessor communication, and storage management. A number of statistics gathered during execution are presented.

Keywords: Compilation, Serial Combinators, Scheduling.

[Goldberg, 1988]
Benjamin F. Goldberg.
Multiprocessor Execution of Functional Programs.
Research report yaleu/dcs/rr-618, Department of Computer Science, Yale University, April 1988.

Abstract: Functional languages have recently gained attention as vehicles for programming in a concise and elegant manner. In addition, it has been suggested that functional programming provides a natural methodology for programming *multiprocessor* computers. This dissertation demonstrates that multiprocessor execution of functional programs is feasible, and results in a significant reduction of their execution times. Two implementations of the functional language ALFL were built on commercially available multiprocessors. *Alfala* is an implementation on the Intel iPSC hypercube multiprocessor, and *Buckwheat* is an implementation on the Encore Multimax shared-memory multiprocessor. Each implementation includes a compiler that performs *automatic decomposition* of ALFL programs. The compiler is responsible for detecting the inherent parallelism in a program, and decomposing the program into a collection of tasks, called *serial combinators*, that can be executed in parallel. One of the primary goals of the compiler is to generate serial combinators exhibiting the coarsest granularity possible without sacrificing useful parallelism. This dissertation describes the algorithms used by the compiler to analyze, decompose, and optimize functional programs. The abstract model supported by Alfala and Buckwheat is called *heterogeneous graph reduction*, which is a hybrid of graph reduction and conventional stack-oriented execution. This model supports parallelism, lazy evaluation and higher order functions while at the same time making efficient use of the processors in the system. The Alfala and Buckwheat run-time systems support dynamic load balancing, interprocessor communication (if required), and storage management. A large number of experiments were performed on Alfala and Buckwheat for a variety of programs. The results of these experiments, as well as the conclusions drawn from them, are presented.

Keywords: Parallelization, Serial Combinators, Graph Reduction.

[Goldman and Gabriel, 1988a]
Ron Goldman and Richard P. Gabriel.
Preliminary Results with the Initial Implementation of Qlisp.
In *ACM Conference on Lisp and Functional Programming*, pages 143–152, Snowbird, Utah, July 1988. ACM Press, New York
Keywords: Lisp, Parallelization.

[Goldman and Gabriel, 1988b]
Ron Goldman and Richard P. Gabriel.
Qlisp: Experience and New Directions.
In *PPEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems*, number 23(9) in SIGPLAN Notices, pages 111–123, New Haven, Conneticut, July 19–21, September 1988. ACM Press, New York.

Abstract: Qlisp, a dialect of Common Lisp, has been proposed as a multiprocessing programming language which is suitable for studying the styles of parallel programming at the medium-grain level. An initial version of Qlisp has benn implemented on a multiprocessor and a number of experiments with it conducted. This paper describes the implementation, reports on some of the experiments, and presents some new constructs that ae suggested from programming experience with Qlisp.

Keywords: Lisp, Futures, Parallel Let.

[Goldman and Gabriel, 1989]
Ron Goldman and Richard P. Gabriel.
Qlisp: Parallel Processing in Lisp.
IEEE Software, pages 51–59, July 1989.

Abstract: There is a serious need for a powerful understandable multiprocessing language. Qlisp is an extension to Common Lisp designed for parallel symbolic computation.

Keywords: Lisp, Futures, Parallel Let.

[Goldman *et al.*, 1988]
Ron Goldman, Richard P. Gabriel, and Carol Sexton.
Qlisp: An Interim Report.
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 161–181, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

Abstract: One of the major problems in writing programs to take advantage of parallel processing has been the lack of good multiprocessing languages — ones which are both powerful and understandable to programmers. In this paper we describe multiprocessing extensions to Common Lisp designed to be suitable for studying styles of parallel programming at the medium-grain level in a shared memory architecture. The resulting language is called Qlisp. A problem with parallel programming is the degree to which the programmer must explicitly address synchronization problems. Two new approaches to this problem look promising: the first is the concept of heavyweight futures, and the second is a new type of function called a partially, multiply invoked function.

Keywords: Lisp, Futures.

[Grit, 1990]
Dale H. Grit.
Sisal on a Message Passing Architecture.
In H. Burkhart, editor, *CONPAR 90 — VAPP IV, Joint International Conference on Vector and Parallel Processing*, volume 457 of *Lecture Notes in Computer Science*, pages 721–731, Zurich, Switzerland, September 10–13, 1990. Springer, Berlin.

Abstract: SISAL is a general-purpose applicative language intended for use on both conventional and novel multiprocessor systems. In this paper we describe the port of a shared memory implementation to a message passing environment. The issues underlying the main features of this implementation are discussed.

Keywords: Sisal, Dataflow.

[Gross and Sussman, 1987]
Thomas Gross and Alan Sussman.
Mapping a Single-Assignment Language onto the Warp Systolic Array.
In Gilles Kahn, editor, *FPCA '87, Functional Programming Language and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 347–363, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

Abstract: Single-assignment languages offer the potential to efficiently program parallel processors. This paper discusses issues that arise in mapping SISAL programs onto the Warp array, a linear systolic array in use at Carnegie Mellon. A Warp machine with ten cells can deliver up to 100 million floating point operations per second. The paper begins with a discussion of systolic arrays as targets for single-assignment languages and the suitability of the Warp machine for this purpose. Systolic arrays can take advantage of both large-grain parallelism and fine-grain parallelism. The communication bandwidth of the systolic array gives the translator great flexibility in mapping a SISAL program onto the linear array. We present two principal methods to exploit parallelism on Warp, data partitioning and pipelining. Data partitioning is effective for local computations that depend on only a small neighborhood of values. Since SISAL allows the specification of array sizes at run-time, we have to provide static and dynamic methods for dynamic data partitioning. Pipelining allows the overlapping of different states of a computation or of function invocations. This method is well suited for Warp since the systolic array has high inter-cell communication bandwidth. This makes it possible to send large data sets to the next processor in a computation pipeline without performance degradation. We use matrix multiplication and a relaxation algorithm, respectively, as examples to illustrate the data partitioning and pipeline models for mapping SISAL programs onto the Warp array.

Keywords: Dataflow, SISAL, SIMD.

[Halstead, Jr., 1985]
Robert H. Halstead, Jr.
Multilisp: A Language for Concurrent Symbolic Computation.
ACM Transactions on Programming Languages and Systems, 7(4):501–538, October 1985.

Abstract: Multilisp is a version of the Lisp dialect Scheme extended with constructs for parallel execution. Like Scheme, Multilisp is oriented toward symbolic computation. Unlike some parallel programming languages, Multilisp incorporates constructs for causing side effects and for explicitly introducing parallelism. The potential complexity of dealing with side effects in a parallel context is mitigated by the nature of the parallelism constructs and by support for abstract data types: a recommended Multilisp programming style is presented which, if followed, should lead to highly parallel, easily understandable programs. Multilisp is being implemented on the 32 processor *Concert* multiprocessor; however, it is ultimately intended for use on larger multiprocessors. The current implementation, called *Concert Multilisp*, is complete enough to run the Multilisp compiler itself and has been run on Concert prototypes including up to eight processors. Concert Multilisp uses novel techniques for task scheduling and garbage collection. The task scheduler helps control excessive resource utilization by means of an unfair scheduling policy; the garbage collector uses a multiprocessor algorithm based on the incremental garbage collector of Baker.

Keywords: Lisp, Futures.

[Halstead, Jr., 1986]
Robert H. Halstead, Jr.
Parallel Symbolic Computing.
IEEE Computer, 19(8):35–43, August 1986.

Abstract: Programs differ from one another in many dimensions. In one such dimension, programs can be laid out along a spectrum with predominantly symbolic programs at one end and predominantly numerical programs at the other. The difference between numerical and symbolic programs suggest different approaches to parallel processing. This article explores the problem and opportunities of parallel symbolic computing and describes the language Multilisp, used at M.I.T. for experiments in parallel symbolic programming.

Keywords: Multilisp, Symbolic Computation.

[Halstead, 1984]
Robert H. Halstead.
Implementation of Multilisp: Lisp on a Multiprocessor.
In *1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, Texas, August 6–8, 1984. ACM Press, New York.

Abstract: Multilisp is an extension of Lisp (more specifically, of the Lisp dialect Scheme) with additional operators and additional semantics to deal with parallel execution. It is being implemented on th 32-processor *Concert* multiprocessor. The current implementation is complete enough to run the Multilisp compiler itself, and has been run on Concert prototypes including up to four processors. Novel techniques are used for task scheduling and garbage collection. The task scheduler helps control excessive resource utilization by means of an unfair scheduling policy: the garbage collector uses a multiprocessor algorithm modeled after the incremental garbage collector of Baker. A companion paper discusses language design issues relating to Multilisp.

Keywords: Multilisp.

[Halstead, 1986]
Robert H. Halstead.
An Assessment of Multilisp — Lessions from Experience.
International Journal of Parallel Programming, 15(6), December 1986
Keywords: Multilisp.

[Halstead, 1987]
Robert H. Halstead.
Overview of Concert Multilisp: A Multiprocesor Symbolic Computation System.
ACM Computer Architecture News, 15(1):5–14, March 1987
Keywords: Multilisp.

[Halstead, 1988]
Robert H. Halstead.
New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools.
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 2–57, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

Abstract: A Lisp-based approach is attractive for parallel computing since Lisp languages and systems assume significant clerical burdens, such as storage management. Parallel Lisps thus enable programmers to focus on the new problems introduced by using concurrency. Parallel Lisps now exist that can execute realistic applications with "industrial-strength" performance, but there are applications whose requirements they do not handle elegantly. Recent Work has contributed new, elegant ideas in the area of speculative computation, continuations, exception handling, aggregate data structures, and scheduling. Using these ideas, it should be possible to build "second generation" parallel Lisp systems that are as powerful and elegantly structured as sequential Lisp systems. This paper surveys these recent ideas and explores how they could fit together in the parallel Lisp systems of the future, examining issues at three levels: language design, implementation techniques, and programming tools. The discussion is based on the Multilisp programming language, which is Scheme (a Lisp dialect) extended with the `future` construct. The paper outlines three criteria for judging Scheme extensions for parallel computing: compatibility with sequential Scheme, invariance of the result when `future` is introduced into side-effect-free Scheme programs, and modularity. Proposed language mechanisms, such as support of first-class continuations, are evaluated against these criteria. In the area of implementation techniques, results of experiments with lazy task creation, unfair scheduling, and parallel garbage collection are surveyed; some areas that need more investigation, such as scheduler implementation for speculative computing, and interaction between user-level and operating-system schedulers, are identified. Finally, past work in tools to help with the development of Multilisp programs is surveyed, and needs for additional tools is discussed.

Keywords: Lisp, Futures.

[Hammarlund and Lisper, 1993]
Per Hammarlund and Björn Lisper.

On the Relation between Functional and Data Parallel Programming Languages.
In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 210–219, Copenhagen, Denmark, June 9–11, 1993. ACM Press.

Abstract: Data parallel programming is becoming an increasingly important tool for exploiting parallelism in data-intensive applications, especially on SIMD and vectore computers. Many algorithms appearing in such applications are very succinctly expressed in data parallel languages: this indicates that data parallel programming can be a powerful abstract programming paradigm than just a syntax for explict programming of SIMD computers. The data parallel languages in practical use today are, however, exponents of exactly the later point of view: even though they incorporate some elements of abstraction, their semantics are all to some extent based on a SIMD execution model. Therefore it is hard to use these languages to express algorithms in the problem domain in an abstract, machine-independent way. This is likely to make programming in these languages more error-prone and programs less portable than if they had been designed with a more clean-cut abstract semantics. Here, we present some mathematical defininitions of data parallel primitives that can be used to guide the design of data parallel languages with a higher level of abstraction. The key idea is to view data parallel entities as tabulated functions where the tables are stored in a distributed fashion. Operations on data parallel entities are then simply operations on functions, just as operations in pure functional languages. An interesting obervation is that also traditional data structures, like lists and arrays, are covered by our view. This illustrates the level of abstraction achieved. An especially interesting possibility is to integrate data parallel and higher order functional languages. Data parallel entities are then just a particular class of functions that can be represented in a particular way. We believe that such languages are very suitable as specification languages for data parallel algorithms.

Keywords: Data Parallelism.

[Hammer and Henties, 1990]
Carsten Hammer and Thomas Henties.
Parallel Lisp for a Distributed Memory Machine.
In *EUROPAL workshop on High Performance and Parallel Computing in Lisp*, Twickenham, UK, November 1990
Keywords: Lisp.

[Hammond and Peyton Jones, 1991]
Kevin Hammond and Simon Peyton Jones.
Profiling scheduling strategies on the GRIP parallel reducer.
Technical report, Department of Computing Science, University of Glasgow, UK, August 1991.

Abstract: It is widely claimed that functional languages are particularly suitable for programming parallel computers. A claimed advantage is that the programmer is not burdened with details of task creation, placement, scheduling, and synchronization, these decisions being taken by the system instead. Leaving aside the question of whether a pure functional language is expressive enough to encompass all the parallel algorithms we might wish to program, there remains the question of how effectively the compiler and run-time system map the program onto a real parallel system, a task usually carried out mostly by the programmer. This is the question we address in this paper. We first introduce the system architecture of GRIP, a shared-memory parallel machine supporting an implementation of the functional language HASKELL. GRIP executes functional programs in parallel using compiled supercombinator graph reduction, a form of declarative rule system. We then describe several strategies for run-time resource control which we have tried, presenting comprehensive measurements for their effectiveness. We are particularly concerned with strategies controlling task creation, in order to improve task granularity and minimize communication overheads. This is, so far as we know, one of the first attempts to make a systematic study of task-control strategies in a high-performance parallel functional-language system. GRIP's high absolute performance render these results credible for real applications.

Keywords: Process Scheduling, Load Balancing.

[Hammond, 1989a]
Kevin Hammond.
Error Handling in the Parallel Implementation of a Lazy Functional Language.

Technical Report C89-01, School of Information Systems, University of East Anglia, Norwich, UK, 1989.

Abstract: This paper presents a version of the error value method which allows lazy computation with deterministic semantics for divide-and-conquer parallel evaluation even in the presence of errors.

Keywords: Error Handling.

[Hammond, 1989b]
Kevin Hammond.
Implementing Functional Languages for Parallel Machines.
PhD thesis, School of Information Systems, University of East Anglia, Norwich, UK, 1989.

Abstract: Commencing with the Standard ML language, dialects XSML and PSML are defined which permit parallel evaluation of functional programs. XSML is Standard ML with a novel mechanism for handling exceptions; PSML is a side-effect free version of XSML. A formal semantics for PSML and a translation algorithm from this language into Dactl, a compiler target language based on the theory of graph-rewriting, are presented. The thesis proves that a simplified version of this translation preserves meaning for flat domains, and that the strategy for reduction to normal form is correct. The implementation of standard compilation techniques such as strictness analysis, maximal free sub-expression elision and common sub-expression elimination is considered with respect to Dactl, and problems highlighted. Techniques are also presented for compiling exception-handling correctly in a parallel environment, and for compiling side-effect for a parallel machine. Metrics for performance evaluation obtained using the Dactl reference interpreter are presented.

Keywords: ML, Graph-Rewriting, Strictness Analysis.

[Hankin and Hunt, 1992]
Chris Hankin and Sebastian Hunt.
Approximate Fixed Points in Abstract Interpretation.
In B. Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 219–232, Rennes, France, February 26–28, 1992. Springer, Berlin.

Abstract: Much of the earlier development of abstract interpretation, and its application to imperative programming languages, has concerned techniques for finding fixed points in large (often infinite) lattices. The standard approach in the abstract interpretation of functional languages has been to work with small, finite lattices and this supposedly circumvents the need for such techniques. However, practical experience has shown that, in the presence of higher order functions, the lattices soon become too large (although still finite) for the fixed-point finding problem to be tractable. This paper develops some approximation techniques which were first proposed by Hunt and shows how these techniques relate to the earlier use of widening and narrowing operations by the Cousots.

Keywords: Fixpoints, Abstract Interpretation.

[Hankin *et al.*, 1988]
Chris L. Hankin, L. Burn, Geoffrey, and Simon L. Peyton Jones.
A Safe Approach to Parallel Combinator Reduction (Extended Abstract).
Theoretical Computer Science, 56(1):17–35, 1988, Also: Extended Abstract in ESOP 86, European Symposium on Programming, Saarbrücken, Germany, March, 1986, pages 99–110, Robinet, B. and Wilhelm, R. (editors), volume 213 of Lecture Nots in Computer Science, Springer, Berlin.

Abstract: In this paper we present the results of two pieces of work which, when combined, allow us to go from a program text in a functional language to a parallel implementation of that program. We present techniques for discovering sources of parallelism in a program at compile time, and then show how this parallelism is naturally mapped into a parallel combinator set that we will define. To discover sources of parallelism in a program, we use abstract interpretation. Abstract interpretation is a compile-time technique which is used to gain information about a program that may then be used to optimize the execution of the program. A particular ue of abstract interpretation is the strictness analysis of functional programs. In a language that has lazy semantics, the main potential for parallelism arises in the evaluation of operands of strict operators. A function is strict in an argument if its value is undefined whenever the argument is undefined. If we can use strictness analysis to detect which arguments a function is strict in, we then know that these arguments can be safely evaluated in parallel because this will not affect the lazy semantics. Having identified the sources of parallelism at compile-time it is necessary to communicate these to the run-time system. In the second part of the paper we use an extended set of combinators, including a pair of parallel combinators that achieve this purpose.

Keywords: Strictness Analysis, Combinators, Director Strings.

[Harmer and Boyle, 1989]
Terence J. Harmer and James M. Boyle.
An Optimization to Increase Parallelism in Parallel Pure Lisp Implementations.
Technical report, Argonne National Laboratory, Argonne, Illinois, 1989.

Abstract: In this paper we consider an optimization that may be applied to the Lisp list constructor *cons* when parallel evaluation of function arguments is possible. We present experimental results comparing implementations of a typical function with and without this optimization.

Keywords: Lisp, Lists.

[Harrison III and Ammarguella, 1988]
William Ludwell Harrison III and Zahira Ammarguella.
The Design of Automatic Parallelizers for Symbolic and Numeric Programs.
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 235–253, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

Abstract: Parcel is a compiler and run-time system that automatically parallelizes a scheme program for execution on a shared memory multiprocessor. Parcel's inter-procedural analysis makes it capable of extracting high-level parallelism from complex applications; it introduced a number of program transformations that are especially useful in parallelizing symbolic computations; and its run-time system makes good use of multiple procedure versions to achieve parallelism without undue inefficiency. In this paper we will describe the strengths and weaknesses of Parcel, and outline the design of Miprac in light of our experience with Parcel.

Keywords: Lisp, Parallelization.

[Harrison III, 1986]
W. Ludwell Harrison III.
Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor.
CSRD Report 565, Center of Supercomputing Research and Development, University of Illinois, Urbana, IL, March 1986.

Abstract: The problem of compiling Lisp for efficient evaluation on a large, tightly coupled, shared memory multiprocessor is investigated. A representation for s-expressions which facilitates parallel evaluation is proposed, along with a sequence of transformations, to be applied to the functions comprising a Lisp program, which reveal and exploit parallelism.

Keywords: Lisp, Compilation, Parallelization.

[Harrison III, 1989]
W. Ludwell Harrison III.
The Interprocedural Analysis and Automatic Parallelization of Scheme Programs.
Technical Report CSRD No. 860, UILU-ENG-89-8003, Center of Supercomputing Research and Development, University of Illinois, Urbana, IL, February 1989.

Abstract: Lisp and its descendants are among the most important and widely used of programming languages. At the same time, parallelism in the architecture of computer systems is becoming commonplace. There is a pressing need to extend the technology of automatic parallelization that has become available to Fortran programmers of parallel machines, to the realm of Lisp programs and symbolic computing. In this thesis we present a comprehensive approach to the compilation of Scheme programs for shared-memory multiprocessors. Our strategy has two principal components: *interprocedural analysis* and *program restructuring*. We introduce *procedure strings* and *stack configurations* as a framework in which to reason about interprocedural side-effects and object lifetimes, and develop a system of interprocedural analysis, using abstract interpretation, that is used in the dependence analysis and memory management of Scheme programs. We introduce the transformations of *exit-loop translation* and *recursion splitting* to treat the control structures of iteration and recursion that arise commonly in Scheme programs. We propose an alternative representation for s-expressions that facilitates the parallel creation and access of lists. We have implemented these ideas in a parallelizing Scheme compiler and run-time system, and we complement the theory of our work with "snapshots" of programs during the restructuring process, and some preliminary performance results of the execution of object codes produced by the compiler.

Keywords: Lisp, Parallelization, Abstract Interpretation.

[Harrison, 1987]
Dave Harrison.
RUTH: A Functional Language for Real-Time Programming.
In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87 Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 297–314, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

Abstract: This paper introduces the real-time programming language RUTH which is a functional language based on the lazy language LispKit LISP. RUTH is a semantically pure functional language in which the concept of a process is encapsulated by the more general concept of a function. Communication between processes is modelled by streams of timestamped values. RUTH programs take as input a tree of integers which is lazily evaluated as the program executes to produce node values representing the current time. Taking real-time information as an input data structure allows RUTH programs to define and react to real-time events without loosing referential transparency and allows us to define a formal semantics without the complication of including time information.

Keywords: Real-Time, Process Networks, Streams.

[Hartel *et al.*, 1991]
Pieter Hartel, Hugh Glaser, and John Wild.
On the Benefits of Different Analyses in the Compilation of a Lazy Functional Language.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 123–145, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: Implementations of lazy functional languages have not as yet approached the speed of execution provided by imperative, or even non-lazy functional languages. It has been suggested that the answer to this problem lies with powerful analyses, based on abstract interpretation, such as strictness analyses that can deal with arbitrary data structures and higher-order functions. In this paper, we present a break down of the benefits of a number of analyses, each performed at increasing levels of sophistication, and analyzed for a set of medium-sized functional programs. We conclude that the benefits of important analyses are limited.

Keywords: Static Analysis.

[Haynes and Friedman, 1984]
Christopher T. Haynes and Daniel P. Friedman.
Engines Build Process Abstractions.
In *1984 ACM Symposium on Lisp and Functional Programming*, pages 18–24, Austin, Texas, August 6–8, 1984. ACM Press, New York.

Abstract: *Engines* are a new programming language abstraction for timed preemption. In conjunction with first class continuations, engines allow the language to be extended with a time-sharing implementation of process abstraction facilities. To illustrate engine programming techniques, we implement a round-robin process scheduler. The importance of simple but powerful primitives such as engines is discussed.

Keywords: Continuations, Process Scheduling.

[Haynes *et al.*, 1984]
Christopher T. Haynes, P. Friedman, Daniel, and Mitchell Wand.
Continuation and Coroutines.
In *1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, Texas, August 6–8, 1984. ACM Press, New York.

Abstract: The power of first class continuations is demonstrated by implementing a variety of coroutine mechanisms using only continuations and functional abstractions. The importance of general abstraction mechanisms such as coroutines is discussed.

Keywords: Continuations, Coroutines.

[Henderson, 1980]
Peter Henderson.
Functional Programming: Application and Implementation.
International Series in Computer Science, Prentice Hall, New Jersey, 1980
Keywords: Streams, Process Networks.

[Henderson, 1982]
Peter Henderson.
Purely Functional Operating Systems.
In J. Darlington, Henderson P., and D. A. Turner, editors, *Functional Programming and its Applications*, pages 177–192. Cambridge University Press, 1982
Keywords: Non-Determinism.

[Hickey and Cohen, 1988]
Timothy Hickey and Jacques Cohen.
Automatic Program Analysis.
Journal of the ACM, 35(1):185–220, January 1988.

Abstract: The first part of the paper shows that previous theoretical work on the semantics of probabilistic programs (Kozen) and on the correctness of performance annotated programs (Ramshaw) can be used to automate the average-case analysis of simple programs containing assignments, conditionals, and loops. A performance compiler has been developed using this theoretical foundations. The compiler is described, and it is shown that special cases of symbolic simplifications of formulas play a major role in rendering the system usable. The performance compiler generates a system of recurrence equations derived from a given program whose efficiency one wishes to analyze. This generation is always possible, but the problem of solving the resulting equations may be complex. The second part of the paper presents an original method that generalizes our previous approach and is applicable to functional programs that make use of recursion and complex data structures. Several examples are presented including an analysis of binary tree sort. A key feature of the analysis of such programs is that distributions on complex data structures are represented using attributed probabilistic grammars.

Keywords: Granularity Analysis.

[Hieb and Dybvig, 1990]
Robert Hieb and R. Kent Dybvig.
Continuations and Concurrency.
In *PPoPP '90, Symposium on Principles and Practice of Parallel Programming*, volume 25(3) of *SIGPLAN NOTICES*, pages 128–136, Seattle, Washington, March 14–16, 1990. ACM Press, New York.

Abstract: Continuations have proven to be useful for implementing a variety of control structures, including exception handling facilities and breadth-first searching algorithms. However, traditional continuations are not useful in the presence of concurrency, because the notion of the rest of the computation represented by a continuation does not in general make sense. This paper presents a new type of continuation, called a *process continuation*, that may be used to control tree-structured concurrency. Just as a traditional continuation represents the rest of a computation from a given point in the computation from a given point in the computation, a process continuation represents the rest of a *subcomputation*, or *process*, from a given point in the *subcomputation*. Process continuations allow nonlocal exits to arbitrary points in the process tree and allow the capture of a subtree of a computation as a composable continuation for later use. Even in the absence of multiple processes, the precise control achievable with process continuations makes them more useful than traditional continuations.

Keywords: Continuations.

[Hill, 1993]
Jonathan M. D. Hill.
Data Parallel Haskell: Mixing Old and New Glue.
Technical report, Queen Mary and Westfield College, University of London, UK, 1993.

Abstract: John Hughes argues that the ability to decompose a problem into parts, depends upon the ability to 'glue' those parts together. A functional programming language such ass Haskell provides the user with two powerful kinds of glue: higher order functions and lazy evaluation. Although many data parallel extensions to functional languages exist — Connection Machine Lisp, Paralation Lisp, Parallel EuLisp, and Tuple, they all lack the special glue of a lazy language. This paper describes two data parallel extensions to the lazy functional language Haskell: PODs and POD comprehensions. The enable parallel algorithms to be expressed by decomposing a problem into smaller sequential parts, which are then glued together in parallel. We contrast an implementation of a text searching algorithm and an LL(1) parer, and show that the parallelism within both can be expressed in terms of the same higher order parallel `map` and `scan` functions. In the first example, we decompose the problem into a sub-string algorithm, and a parallel line joining algorithm. We glue these two subproblems together by using lazy evaluation and POD comprehensions — the old and new glue seem to work well.

Keywords: Data Parallelism.

[Hogen and Loogen, 1992a]
Guido Hogen and Rita Loogen.
A New Stack Technique for the Management of Runtime Structures in Distributed Implementation.
Technical report, Lehrstuhl für Informatik II, RWTH Aachen, Germany, 1992.

Abstract: A new technique for the management of runtime structures in distributed implementations is presented. Instead of storing the runtime stacks of parallel processes as linked lists of function blocks in a heap structure, the local stacks of several parallel processes, which are executed on the same processor element, are stored in an interleaved manner on a single physical stack, called the *meshed stack*. The technique originated from the stack organization in the well known Warren Abstract Machine (WAM) for logic languages, where the runtime information of several alternative computation branches is stored on a single stack. In principle, the meshed stack technique is independent of the implemented language. We will explain it for the parallel implementation of functional language.s The new technique supports sequential and parallel evaluations equally well.

Keywords: Parallel Graph Reduction, Stacks.

[Hogen and Loogen, 1992b]
Guido Hogen and Rita Loogen.
PASTEL — A Parallel Stack-based Implementation of Eager Functional Programs with Lazy Data Structures.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: In this paper we present an alternative realization of the runtime structures which tends to support parallel and sequential evaluations as well. Instead of representing the cactus stack as a linked tree of function blocks, we allocate function blocks, within each processor element of the parallel system, on a single *meshed stack*, which is organized as a contiguous array, but shared by several processes. The graph is only used for the representation of parallel processes, structured data, closures and suspensions. Evaluation is controlled by the stack. During the sequential evaluation of processes the meshed stack is used as in conventional implementations. Function blocks aer allocated on top of it, when function calls are executed, and removed, when function calls terminate.

Keywords: Parallel Graph Reduction, Stacks.

[Hogen *et al.*, 1992]
Guido Hogen, Andrea Kindler, and Rita Loogen.
Automatic Parallelization of Lazy Functional Programs.
In B. Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 254–268, Rennes, France, February 26–28, 1992. Springer, Berlin.

Abstract: We present a parallelizing compiler for lazy functional programs that uses strictness analysis to detect the implicit parallelism within programs. It generates an intermediate functional program, where a special syntactic construct *'letpar'*, which is semantically equivalent to the well-known *let*-construct, is used to indicate subexpressions for which a parallel execution is allowed. Only for sufficiently complex expressions a parallelization will be worthwile. For small expressions the communication overhead may outweigh the benefits of the parallel execution. Therefore, the parallelizing compiler uses some heuristics to estimate the complexity of expressions. The distributed implementation of parallelized functional programs enabled us to investigate the impact of various parallelization strategies on the runtimes and speedups. The strategy, which only allows the parallel execution of non-predefined function calls in strict positions shows the best runtimes and reasonable speedup results.

Keywords: Compilation, Strictness Analysis, Evaluation Transformers.

[Howe and Burn, 1992]
Denis B. Howe and L. Burn, Geoffrey.
Experiments with Strict STG Code.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: Lazy evaluation has many desirable features for the functional programmer but imposes certain overheads on the execution mechanism. Much work has gone into making lazy evaluation efficient. One of the most promising projects in this area is GHC, Glasgow University's Haskell compiler based on the Spineless Tagless G-Machine. Strictness analysis tells us where we can turn lazy evaluation into strict evaluation without changing the results of the program. Much theoretical work has gone into making strictness analysis more powerful and efficient enough for practical use. Less work has focussed on using the results of strictness analysis to generate better code and it is this that we consider here. We have taken the results of one kind of strictness analysis, known as evaluation transformers, and used them to transform the lazy code produced by GHC to make it more strict. We describe the execution of lazy and strict code by the STG Machine and explain why the strict code is not always faster. Finally, we suggest alternative code generation strategies and program transformations which might improve the strict code.

Keywords: Strictness Analysis, Evaluation Transformers, Code Generation.

[Huang *et al.*, 1991]
Shell-Ying Huang, Paul Kelly, and Junxian Liu.
Pilot Implementation of Abstract, Declarative Process Placement.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 197–214, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: To get high performance on a distributed-memory multicomputer some explicit control is needed, at present and for the foreseeable future. This paper describes work aimed at harnessing the power of the functional notation in exercising such control. We have developed a declarative annotation scheme which allows explicit control over process placement and communications. The language, called Caliban, is being implemented on a configurable, loosely-coupled commercial multicomputer and a pilot version is described here.

Keywords: Para-Functional Programming, Annotations.

[Hudak and Anderson, 1987]
Paul Hudak and Steve Anderson.
Pomset Interpretations of Parallel Functional Programs.
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 234–256, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

Abstract: A new framework is presented, based on the notion of a *partially ordered multi-set* (or *pomset*), which is able to provide not only a precise operational semantics of parallel functional program evaluation, but also a handle through which to control such behavior. As an operational semantics, pomsets are able to distinguish between call-by-value, call-by-name, call-by-need, and call-by-speculation evaluation strategies (even though all but the first of these have the same standard semantics); and as a "handle" from which to control operational behavior, pomsets can express most of the behaviors achieved by previously proposed annotations that control not only evaluation order but also the spatial mapping of program to machine.

Keywords: Semantics, Speculative Parallelism.

[Hudak and Anderson, 1988]
Paul Hudak and Steve Anderson.
Haskell Solutions to the Language Session Problems at the 1988 Salishan High-Speed Computing Conference.
Research Report YALEU/DCS/RR-627, Department of Computer Science, Yale University, May 1988.

Abstract: This paper is intended to give the reader some familiarity with Haskell by giving solutions to the four language session problems presented at the 19888 Salishan Conference on High-Speed Computing.

Keywords: Functional Programming.

[Hudak and Goldberg, 1984]
Paul Hudak and Benjamin Goldberg.
Experiments in Diffused Combinator Reduction.
In *1984 ACM Symposium on Lisp and Functional Programming*, pages 167–176, Austin, Texas, August 6–8, 1984. ACM Press, New York.

Abstract: In recent years there has been a fair amount of interest both in using *combinators* to represent functional programs, and in using *graph reduction* as an underlying evaluation strategy. Combining these ideas within a single framework for an "applicative architecture" is very appealing because: (1) the normally ubiquitous "environment" is eliminated, (2) the evaluation strategy becomes very simple (amenable to VLSI), and (3) there is a great potential for parallelism. We have been exploring a model of diffused combinator reduction in which the reduction process is distributed "by demand" among a network of closely-coupled processors. We have tested our ideas via simulation, with encouraging results.

Keywords: Parallel Graph Reduction, Combinators.

[Hudak and Goldberg, 1985a]
Paul Hudak and Benjamin Goldberg.
Distributed Execution of Functional Programs Using Serial Combinators.
IEEE Computer, 34(10):881–891, October 1985.

Abstract: A general strategy for automatically decomposing and dynamically distributing a functional program is discussed, suitable for parallel execution on multiprocessor architectures with no shared memory. The strategy borrows ideas from data flow and reduction machine research on one hand, and from conventional compiler technology for sequential machines on the other. One of the more troublesome issues in such a system is choosing the right granularity for the parallel tasks. As a solution we describe a program transformation technique based on *serial combinators* that offers in some sense just the "right" granularity for this style of computing, and that can be "fine-tuned" for particular multiprocessor architectures. We show via simulation the success of our approach.

Keywords: Compilation, Serial Combinators, Load Balancing.

[Hudak and Goldberg, 1985b]
Paul Hudak and Benjamin Goldberg.
Serial Combinators: Optimal Grains of Parallelism.
In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 382–399, Nancy, France, September 16–19, 1985. Springer, Berlin.

Abstract: A method is described for translating a high-level functional program into combinators suitable for execution on multiprocessors with no shared memory. It is argued that the granularity of the standard set of fixed combinators is too fine, whereas the granularity of user-defined functions is too coarse. The notion of a *serial combinator* is introduced that in some sense has optimal granularity, and that takes into account pragmatic issues such as the complexity of expressions and communication costs between processors. The technique improves on the standard notion of *super-combinators* by making them larger to retain locality and improve efficiency, and by ensuring that they have no concurrent substructure that could result in lost parallelism. Simulation results demonstrate improved performance on both sequential and parallel computing models.

Keywords: Parallelization, Serial Combinators.

[Hudak and Smith, 1986]
Paul Hudak and Lauren Smith.
Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems.
In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 243–254, St. Petersburg Beach, Florida, January 15–16, 1986. ACM Press, New York.

Abstract: One of the most important pragmatic advantages of functional languages is that concurrency in a program is *implicit* — there is no need for special constructs to express parallelism as is required in most conventional languages. Furthermore, it is fairly easy for compilers to automatically determine the concurrency as a step toward decomposing a program for execution on a suitable parallel architecture. Yet it is often the case that one knows precisely the *optimal decomposition* for execution on a particular machine, and one can never expect a compiler to determine such optimal mappings in all cases. This paper is concerned with ways to allow the programmer to *explicitly* express this mapping of program to machine, by using annotations that, given a few minor constraints, cannot alter the functional semantics of the program. We show through several detailed examples the expressiveness and conciseness of the resulting "para-functional" programming methodology, using an experimental language called *ParAlfl* based on our ideas. We also give a formal denotational description of the mapping semantics using a notion of *execution trees*.

Keywords: Para-Functional Programming, Annotations, Process Mappings.

[Hudak and Sundaresh, 1988]
Paul Hudak and Raman S. Sundaresh.
On the Expressiveness of Purely Functional I/O Systems.
Research Report YALEU/DCS/RR-665, Department of Computer Science, Yale University, December 1988.

Abstract: Functional programming languages have traditionally lacked complete, flexible, and yet referentially transparent I/O mechanisms. Previous proposals for I/O have used either the notion of *lazy streams* or *continuations* to model interaction with the external world. We discuss and generalize these models and introduce a third, which we call the *systems* model, to perform I/O. The expressiveness of the styles are compared by means of an example. We then give a series of surprisingly simple translations between the three models, demonstrating that they are not as different as their programming styles suggest, and implying that the styles could be mixed within a single program. The need to express non-deterministic behavior in a functional language is well recognized. So is the problem of doing so without destroying referential transparency. We survey past approaches to this problem, and suggest a solution in the context of the I/O models described. The I/O system of the purely functional language Haskell is presented. The system includes a rich set of operations, and distinguishes between *file* and *channel* I/O. The approach to non-determinism is also presented. A useful aspect of the design is that it includes a rigorous specification of the behavior of the operating system, thus precisely fixing the semantics of the various I/O operations. The Haskell I/O system is capable of supporting may other paradigms of concurrent computation in a natural way. We demonstrate this through the emulation of Actors, UNITY, CSP, CCS and Linda.

Keywords: Streams, Continuations, Non-determinism.

[Hudak and Young, 1986]
Paul Hudak and Jonathan Young.
Higher-Order Strictness Analysis in Untyped Lambda Calculus.

In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 97–109, St. Petersburg Beach, Florida, January 15–16, 1986. ACM Press, New York.

Abstract: A function is said to be *strict* in one of its formal parameters if, in all calls to the function, either the corresponding actual parameter is evaluated, or the call does not terminate. Detecting which arguments a function will surely evaluate is a problem that arises often in program transformation and compiler optimization. We present a strategy that allows one to infer strictness properties of functions expressed in the lambda calculus. Our analysis improves on previous work in that (1) a set-theoretic characterization of strictness is used that permits treatment of *free variables*, which in turn permits a broader range of interpretations, and (2) the analysis provides an effective treatment of higher-order functions. We also prove a result due to Meyr: the problem of first-order strictness analysis is complete in deterministic exponential time. However, because the size of most functions is small, the complexity seems to be tractable in practice.

Keywords: Strictness Analysis, Abstract Interpretation.

[Hudak, 1983]
Paul Hudak.
Distributed Task and Memory Management.
In N. A. Lynch et al., editors, *Symposium on Principles of Ditributed Computing.* ACM Press, New York, August 1983.

[Hudak, 1986a]
Paul Hudak.
Arrays, Non-Determinism, Side-Effects, and Parallelism: A Functional Perspective (Extended Abstract).
In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction, Proceedings of a Workshop*, volume 279 of *Lecture Notes in Computer Science*, pages 312–327, Santa Fe, New Mexico, September 29 – October 1, 1986. Springer, Berlin.

Abstract: Purely functional incremental updates to arrays, executed in a non-deterministic manner, are shown to achieve the same effect (in both efficiency and functionability) as parallel assignment to imperative arrays. The strategy depends only on the ability of a compiler to recognize that incremental updates to functional arrays can be done destructively (an optimization often called "copy-avoidance" of "single-threaded arrays".) Special "pseudo-functional syntax" is introduced that captures typical array-update patterns. This syntax falls somewhere between the purely functional and (im)purely imperative, and makes the inferencing problem fairly easy. If nothing else, the work represents an interesting intellectual exercise in the relationship between non-determinism, side-effects, and parallelism, and tends to blur the traditionally clear-cut distinction between functional and imperative languages.

Keywords: Non-Determinism, Arrays.

[Hudak, 1986b]
Paul Hudak.
Denotational Semantics of a Para-functional Programming Language.
International Journal of Parallel Programming, 15(2):103–125, 1986
Keywords: Para-Functional Programming.

[Hudak, 1986c]
Paul Hudak.
Para-Functional Programming.
IEEE Computer, 19(8):60–70, August 1986.

Abstract: In this article I introduce *para-functional programming*, a methodology for programming multiprocessor computing systems. It is based on a functional programming model augmented with features that allow programs to be mapped to specific multiprocessor topologies. The most significant aspect of the methodology is that it treats the multiprocessor as a single autonomous computer onto which a program is mapped, rather than as a group of independent processors that carry out complex communication and require complex synchronization. In more conventional approaches to parallel programming, the latter method of treatment is often manifested as processes that cooperate by message-passing. However, such notions are absent in para-functional programming; indeed, a single language and evaluation model can be used from problem inception, to prototypes targeted for uniprocessors, and ultimately to realization on a parallel machine.

Keywords: Para-Functional Programming, Annotations, Process Mapping.

[Hudak, 1991]
Paul Hudak.
Para-Functional Programming in Haskell.
In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 5, pages 159–196, ACM Press, New York, 1991.

Abstract: In this chapter I will introduce an extension to the functional programming paradigm, called *para-functional programming*, that allows us to express the aforementioned kinds of behavior without losing any of the well-known benefits of functional languages. The idea is based on preserving the qualitative difference between *functional* behavior (that is, *what* the program computes) and *operational* behavior (that is, *how* the answer is computed). The ideas can be adopted in any functional language, but I will express them in *Haskell*, a recently designed functional language that captures most of the major innovations in functional language research.

Keywords: Para-Functional Programming, Process Scheduling, Mapping.

[Hughes and O'Donnell, 1989]
John Hughes and John O'Donnell.
Expressing and Reasoning about Non-deterministic Functional Programs.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 287–295, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

Abstract: In this paper we introduce a new way of expressing non-determinism functionally. Our approach is unsurprising — we simply add sets of possible value as a new datatype — yet it avoids many of the pitfalls of earlier attempts. We are able to prove theorems about non-deterministic programs using equational reasoning, just as we do in the deterministic case.

Keywords: Non-Determinism.

[Hughes, 1985]
John Hughes.
Strictness Analysis in Non-Flat Domains.
In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects, Proceedings of a Workshop*, volume 217 of *Lecture Notes in Computer Science*, pages 112–135, Copenhagen, Denmark, October 17–19, 1985. Springer, Berlin.

Abstract: In this paper, a strictness analysis method is introduced which works for functions on non-flat domains.

Keywords: Strictness Analysis, Contexts.

[Hughes, 1987a]
R. J. M. Hughes.
Analysing Strictness by Abstract Interpretation of Continuations.
In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 4, Ellis Horwood, 1987
Keywords: Strictness Analysis, Abstract Interpretation, Continuations.

[Hughes, 1987b]
R. J. M. Hughes.
Backwards Analysis of Functional Programs.
Departmental Research Report CSC/87/R3, Department of Computing Science, University of Glasgow, Scotland, 1987.

Abstract: This paper outlines a higher-order method for strictness analysis: it seems to be relational in its forward (higher-order) components, and independent in its backward (first-order) components. In fact, the construction is inconsistent. This method is not based on projections, but a rather more general concept of contexts, but involves essentially the same kind of abstract domains.

Keywords: Strictness Analysis, Contexts.

[Hughes, 1989]
R. J. M. Hughes.
Projections for Polymorphic Strictness Analysis.
In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigne, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 82–100, Manchester, September 5–8, 1989. Springer, Berlin.

Abstract: We apply the categorical properties of polymorphic functions to compile-time analysis, specifically projection-based strictness analysis. First we interpret parameterized types as functors in a suitable category, and show that they preserve monics and epics. Then we define "strong" and "weak" polymorphism — the latter admitting certain projections that are not polymorphic in the usual sense. We prove that, under the right conditions, a weakly polymorphic function is characterized by a single instance. It follows that the strictness analysis of one simple instance of a polymorphic function yields results that apply to all. We show how this theory may be applied. In comparison to earlier polymorphic strictness analysis methods, ours can apply polymorphic information to a particular instance very simply. The categorical approach simplifies our proofs enabling them to be carried out at a higher level, and making them independent of the precise form of the programming language to be analyzed.

Keywords: Strictness Analysis, Projections, Category Theory.

[Hughes, 1990]
John Hughes.
Compile-time Analysis of Functional Programs.
In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 5, pages 117–154, Addison-Wesley, Reading, Massachusetts, 1990.

Abstract: Good compilers for functional languages increasingly rely on sophisticated compile-time analysis to detect opportunities for optimization. As a result, techniques for performing these analyses are developing very rapidly. In this paper we give an informal introduction to two kinds of analysis technique: forwards and backwards analysis. Forwards analysis, or *abstract interpretation*, essentially consists of running the program with partial information about its inputs, to derive partial information about its result. This technique has been widely used for a number of different analysis problems. Backwards analysis is used to derive contextual information by "running the program backwards": It has an efficiency advantage. We will show how the techniques are applied to first-order functional languages, taking strictness analysis as our principal example, and then we will go on to describe three other applications of backwards analysis. We show how backwards analysis can be extended to analyze data structures. Finally we give some pointers to related work.

Keywords: Strictness Analysis, Abstract Interpretation, Backwards Analysis.

[Hunt and Hankin, 1991]
Sebastian Hunt and Chris Hankin.
Fixed Points and Frontiers: A New Perspective.
Journal of Functional Programming, 1(1):91–120, January 1991.

Abstract: Abstract interpretation is the collective name for a family of semantics-based techniques for compile-time analysis of programs. One of the most costly operations in automating such analyses is the computation of fixed points. The frontiers algorithm is an elegant method, invented by Chris Clack and Simon Peyton Jones, which addresses this issue. In this article we present a new approach to the frontiers algorithm based on the insight that frontiers represent upper and lower subsets of a function's argument domain. This insight leads to a new formulation of the frontiers algorithm for higher-order functions, which is considerably more concise than previous versions. We gon on to argue that for many functions, especially in the higher-order case, finding fixed points is an intractable problem unless the sizes of the abstract domains are reduced. We show how the semantic machinery of abstract interpretation allows us to place upper and lower bounds on the values of fixed points in large lattices by working within smaller ones.

Keywords: Abstract Interpretation, Fixed Points, Frontiers.

[Hunt, 1989]
Sebastian Hunt.
Frontiers and Open Sets in Abstract Interpretation.
In *FPCA '89 Functional Programming Languages and Computer Architecture*, pages 1–11, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: Abstract interpretation is the name applied to a range of techniques which use non-standard semantics to identify properties of computer programs. The central task in applying these techniques is to find fixed points of recursive definitions. To fulfill this task efficiently, a compact representation of function values is needed. Clack and Peyton Jones showed how sets of incomparable points from a function's argument domain, known as *frontiers*, could meet this need for function spaces of the form $[2^n \to 2]$. They also developed the *frontiers algorithm*, a method of establishing the frontier representation of a function. Martin and Hankin extended the method to cope with higher-order functions over more general finite lattices. In this paper we present a new approach to the frontiers algorithm based on the insight that frontiers represent open and closed subsets of a function's argument domain. This insight leads to a new formulation of the frontiers algorithm for higher-order functions over a rich family of finite lattices. This formulation is considerably more concise than previous versions. We go on to argue that for many functions, especially in the higher-order case, finding fixed points is an intractable problem unless the sizes of the abstract domains are reduced. We show how the semantic machinery of abstract interpretation allows us to place upper and lower bounds on the values of fixed points in large lattices by working within smaller ones.

Keywords: Abstract Interpretation, Fixed Points, Frontiers.

[Hurson and Lee, 1989]
A. R. Hurson and B. Lee.
Hybrid Structure: A Scheme for Handling Data Structuress in a Data Flow Environment.
In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, volume 365 of *Lecture Notes in Computer Science*, pages 323–340, Eindhoven, The Netherlands, June 12–16, 1989. Springer, Berlin.

Abstract: The asynchronous nature of data flow model of computation allows the exploitation of maximum inherent parallelism in many application programs. However, before data flow model of computation can become a viable alternative to control flow model of computation, one has to find practical solutions to some major problems such as efficient handling of data structures. This article introduces a new model for handling data structures in a data flow environment. The proposed model combines both aspects of copying and sharing to optimize the storage and processing overhead incurred during the operation on data structures. In addition, using simulation, a comparative analysis of our model against other data structure models proposed in literature is presented.

Keywords: Distributed Data Structures.

[Hwang and Rushall, 1992]
Suntae Hwang and David Rushall.
The ν-STG machine: A Parallelized Spineless Tagless Graph Reduction Machine in a Distributed Memory Architecture (Draft Version).
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: This paper describes the ν-STG machine a parallelized Spineless Tagless Graph Reduction (STG) machine in a distributed memory architecture. In the ν-STG machine, a stack for each task is distributed by allocating a *context frame* for each *tail-call sequence* on the *heap*. Two sparking mechanisms, *BSpark* and *ISpark*, are supported to introduce parallelism, which are similar to those in the HDG machine. The *BSpark* cannot be ignored and should be synchronized with the parent task to prevent it being blocked and resumed too frequently, while *ISpark* may be ignored. Even though a variable is sparked by *BSpark* or *ISpark*, it may be evaluated *in-line* by the parent task. Creating a new task to evaluate a *boxed* variable according to the spark annotation is delayed until it is really needed. A message passing mechanism is used to distribute jobs and synchronize them in a machine. A *lazy task creation mechanism* is supported to exploit paallelism in *unboxed* arithmetic expression by *boxing up* on demand. A simple printing mechanism is added to print outputs clearly.

Keywords: Parallel Graph Reduction, G-Machine.

[Iannucci, 1988]
Robert Alan Iannucci.
A Dataflow/von Neumann Hybrid Architecture.
Technical Report MIT/LCS/TR-418, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1988.

Abstract: Dataflow architectures offer the ability to trade program-level parallelism in order to overcome machine-level latencies in accessing memory and in communicating with other processors. Dataflow further offers a uniform synchronization paradigm, representing one end of spectrum wherein the unit of scheduling is a single instruction. At the opposite extreme are the von Neumann architectures which schedule on a task, or process, basis. As a basis for scalable, general purpose multiprocessors, traditional von Neumann architectures are unsuitable due to their inability to tolerate latency and to provide means for fine-grained synchronization. This report examines the spectrum by proposing a new architecture which is a *hybrid* of dataflow and von Neumann organizations. The analysis attempts to discover those features of the dataflow architecture, lacking in a von Neumann machine, which are essential for tolerating latency and synchronization costs. These features are captured in the concept of a *parallel machine language* which can be grafted on top of an otherwise traditional von Neumann base. In such an architecture, the units of scheduling, called *scheduling quanta*, are bound at compile time rather than at instruction set design time. The parallel machine language supports this notion via a large synchronization anme space. It is shown that the combination of dataflow-style explicit synchronization and von Neumann-style implicit synchronization in the same instruction set results in an architectural synergism. Using an instruction set which is strictly less powerful than that of the MIT Tagged-Token Dataflow Architecture (TTDA), the hybrid architecture can exploit the same kinds of parallelism than the TTDA. Given that a compiler can generate scheduling quanta of two or three instructions, the hybrid architecture will execute approximately the same number of instructions as the TTDA. Larger quanta can result in the hybrid actually executing *fewer* instructions than the TTDA, demonstrating the power of passing state implicitly between program-counter sequenced instructions.

Keywords: Dataflow, Hybrid, Compilation.

[Ilmberger Hermann, 1991]
Sabine Ilmberger Hermann, Thürmel.
A Toolkit for Debugging Parallel Lisp Programs.
In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE '91, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms*, volume 505 of *Lecture Notes in Computer Science*, pages 406–422, Eindhoven, The Netherlands, June 10–13, 1991. Springer, Berlin.

Abstract: As part of the ESPRIT-II project EDS a toolkit (called Delphi) is under development for the debugging of Lisp programs with explicit parallelism being executed on a homogeneous distributed machine. It assists in the detection of functional and synchronisation errors. It also helps to detect unexpected nondeterminacy and sources of poor program performance. Specific mechanisms allow the user to effectively control several processes in a debugging session. The paper introduces the basic concepts behind the tools and how the user may benefit from them.

Keywords: Lisp, Debugging.

[Ito and Matsu, 1988]
Takayasu Ito and Manabu Matsu.
A Parallel Lisp Language PaiLisp and its Kernel Specification.
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 58–100, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

Abstract: Parallel Lisp languages have been proposed and developed for parallel programming of Lisp programs in Artificial Intelligence and Symbolic computing. PaiLisp is a parallel Lisp language based on a shared-memory parallel architecture. PaiLisp has been designed so as to include (1) parallel constructs such as `future` and `pcall` of Multilisp, and `exlambda` for mutual exclusion and `spawn` and `par` for process creation and (2) parallelized Lisp constructs such as parallel conditional expression, parallel mapcar, parallel and/or etc. The first parallel version of PaiLisp, designed in 1986, was based on FranzLisp and the next version was based on Common Lisp. The current version of PaiLisp is based on Scheme. That is, the current version of PaiLisp may be viewed as an extension of Scheme with parallel Lisp constructs mentioned above. In this paper we present (1) the language specification of PaiLisp, (2) PaiLisp-Kernel, which is a compact kernel of PaiLisp, (3) Description of PaiLisp using PaiLisp-Kernel.

Keywords: Lisp, Futures.

[Iwasaki, 1988]
Hideya Iwasaki.
mUtilisp: a Lisp Dialect for Parallel Processing.
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 316–321, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

Abstract: The mUtilisp (multiprocessing Utilisp) language is a Lisp dialect which allows multiprocessing computation. since it is an extension of the Utilisp (University of Tokyo Interactive Lisp) system, almost all functions of Utilisp are implemented with the same meaning in the mUtilisp system. This paper proposes an aproach to the multiprocessing carried out in the mUtilisp system.

Keywords: Lisp, Processes.

[Jagannathan, 1986]
Suresh Jagannathan.
A Model of Data Backup and Recovery in a Computer System for Functional Programming.
In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, Ebberup, Denmark, August 25–28, 1986. North-Holland, Amsterdam.

Abstract: We present a formal operational model of an abstract data flow interpreter for an application programming language. This model serves as a precise specification of an idealized implementation for the VIM computer system, an experimental system under development at MIT intended to examine the efficient implementation of functional languages using data flow principles. The correctness of any implementation of VIM can be verified by showing that it preserves all the invariants specified by this abstract model. To demonstrate this technique, we present an extension of this idealized system which contains backup and recovery facilities to guarantee the security of all online data against loss or corruption as a result of hardware malfunction. To show that the augmented system correctly preserves the system state, we use as simple inductive proof technique that establishes the equivalence of the behavior of the two systems. Our specification language is a simple applicative language augmented with operations to perform first-order existential and universal quantification.

Keywords: Operating System, Streams, Operational Model.

[Jayaraman and Plaisted, 1987]
Bharat Jayaraman and David A. Plaisted.
Functional Programming with Sets.
In Gilles Kahn, editor, *FPCA '87, Functional Programming Language and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 194–211, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

Abstract: Set abstraction, originally introduced in functional languages by Turner, is an appealing construct because it leads to concise definitions of many interesting operations. However, existing approaches treat sets as lists for the sake of efficiency, and thereby sacrifice a simple declarative semantics. In this paper, we present a novel language based on sets and equations, where sets are treated as sets, consistent with their semantics. The language is called SEL, for Set-Equational Language. Equations are assumed to define a confluent rewriting system when oriented left to right. Sets are defined in terms of their subsets; these rules define a nonconfluent rewriting system when oriented left to right. We show examples of programs in this language, and provide an operational semantics for such programs. Programs are executed by innermost reduction, which may be nondeterministic or deterministic. Nondeterministic reduction is used when one of the elements of a set is desired. Deterministic reduction is used to simplify a term via an equation or to obtain all the elements of a set. The correctness of the operational semantics is also established.

Keywords: Non-Determinism, Sets.

[Jensen, 1991]
Thomas P. Jensen.
Strictness Analysis in Logical Form.
In John Hughes, editor, *Functional Programming Languages and Computer Architectures*, volume 523 of *Lecture Notes in Computer Science*, pages 352–366, Harvard, Massachusetts, USA, 1991. Springer, Berlin.

Abstract: This paper presents a framework for comparing two strictness analysis techniques: Abstract interpretation and non-standard type inference. The comparison is based on the representation of a lattice by its ideals. A formal system for deducing inclusions between ideals of a lattice is presented and proved sound and complete. Viewing the ideals as *strictness properties* we use the formal system to define a program logic for deducing strictness properties of expressions in a typed lambda calculus. This *strictness logic* is shown to be sound and complete with respect to the abstract interpretation, which establishes the main result that strictness analysis by type-inference and by abstract interpretation are equally powerful techniques.

Keywords: Strictness Analysis, Abstract Interpretation, Type Inference.

[Johnson, 1985]
Thomas Johnson.
Lambda Lifting: Transforming Programs to Recursive Equations.
In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, France, September 16–19, 1985. Springer, Berlin.

Abstract: Lambda lifting is a technique for transforming a functional program with local function definitions, possibly with free variables in the function definitions, into a program consisting only of global function (combinator) definitions which will be used as rewrite rules. Different ways of doing lambda lifting are presented, as well as reasons for rejecting or selecting the method used in our Lazy ML compiler. A functional program implementing the chosen algorithm is given.

Keywords: Lambda Lifting, Program Transformation.

[Johnsson, 1991]
Thomas Johnsson.
Parallel Evaluation of Functional Programs: The $\langle \nu, G \rangle$ Approach (Summary).
In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE '91, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms*, volume 505 of *Lecture Notes in Computer Science*, pages 1–5, Eindhoven, The Netherlands, June 10–13, 1991. Springer, Berlin.

Abstract: An overview on the abstract parallel graph reduction machine $\langle \nu, G \rangle$ is given.

Keywords: Parallel Graph Reduction.

[Jones and Le Metayer, 1989]
Simon B. Jones and Daniel Le Le Metayer.
A New Method for Strictness Analysis on Non-Flat Domains.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 1–11, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

Abstract: There are two sides of strictness analysis to be investigated: methods for performing the strictness analysis, and exploitation of the results of the analysis. In this paper we shall address the former, in particular in the case of lazily evaluated data structures. Our approach to this problem is novel in that it is based on an abstract domain of very general *necessity patterns* which allow us to model in great detail the requirements that a function has for the components of the data structures which form its arguments, and to generate finite abstract domains in a systematic way (to enable the computation of fixpoints at compile time), and in a way that can be easily customized for the application. We refer to the analysis using necessity patterns as *necessity analysis*. This is essentially a form of *backwards analysis* but carried out in a different framework.

Keywords: Strictness Analysis, Abstract Interpretation, Backwards Analysis.

[Jones and Sinclair, 1989]
S. B. Jones and A. F. Sinclair.
Functional Programming and Operating Systems.
The Computer Journal, 32(2):162–174, April 1989.

Abstract: There is a large class of programming problems to which, at first sight, functional programming does not seem suited: interactive programs and programs which must access external resources such as file stores or communication systems. We take operating systems as the archetype of this class of problems. We show how the lazy evaluation of functional programs, in particular lazily evaluated infinite lists, or streams, can be exploited to write interactive programs. This extends quite naturally to a software design paradigm comprising processes and networks, to the control of peripheral devices, and hence to the design of operating systems. We present a design for a simple, single user, multiprogramming operating system. This demonstrates workable solutions to operating system structuring, to low level device interfacing, to high level application program interfacing, and to resource management. We examine the paradigm illustrated, with reference to alternative approaches which have been proposed.

Keywords: Non-Determinism, Streams, Process Networks, Operating Systems.

[Jones, 1983]
S. B. Jones.
Abstract Machine Support for Purely Functional Operating Systems.
Number PRG-34 in Technical Monograph, Oxford University Press, August 1983
Keywords: Operating System.

[Jones, 1984]
S. B. Jones.
A Range of Operating Systems written in a Purely Functional Style.
Number PRG-42 in Technical Monograph, Oxford University Press, September 1984
Keywords: Operating System.

[Jorrand *et al.*, 1985]
Ph. Jorrand, J-M. Hufflen, A. Marty, J-Ch. Marty, and Ph. Schnoebelen.
FP2 — The Language and its Formal Definition.
Research Report LIFIA 26, Laboratoire d' Informatique Fondamentale et d' Intelligence Artificielle, IMAG Informatique et Mathematiques Appliquees de Grenoble, France, May 1985.

Abstract: The purpose of this report is to provide both an informal introduction to the main concepts of FP2 and a formal definition of its syntax and semantics. *It must be noted that both the language FP2 and its formal definition are still subject to change.* In particular, it is expected that feedback from implementation and early use of FP2 will be the source of modifications to the language.

Keywords: FP2, Process Networks.

[Jorrand, 1984]
Philippe Jorrand.
FP2: Functional Parallel Programming Based on Term Substitution.
Research Report LIFIA 15, Laboratoire d' Informatique Fondamentale et d' Intelligence Artificielle, IMAG Informatique et Mathematiques Appliquees de Grenoble, France, November 1984.

Abstract: The purpose of this paper is to present FP2 (Functional Parallel Programming), a language where communication, functions and parallelism are consistent notions: its design is simple and its formal semantics are rigorously defined. This is made possibleby a systematic use of terms and term substitutions for representing the basic notions of the language.

Keywords: FP2, Process Networks.

[Jouret, 1991]
Guido K. Jouret.
A Foundation for Declarative Data Parallelism.

In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 311–330, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: Data-parallelism is inherent in the independent activity of individual data elements where this activity consists of either communication or computation. Single-Instruction, Multiple Datastream (SIMD) architectures have been designed to exploit this form of parallelism but despite the impressive performance of such architectures, language development for data-parallel architectures is an under-developed area. The main difficulties in language-design for the exploitation of data-parallelism is the traditional reliance on non-monolithic language constructs encouraged by von-Neumann architectures; the uniform and globally-accessible machine state and the resulting absence of any notions of locality and communication. Consideration of these issues and their relevance on language design leads to the development of a declarative language for data-parallel programming based on the functional style.

Keywords: Data-Parallelism, SIMD, Para-Functional Programming.

[Joy, 1990]
M. S. Joy.
Parallel Evaluation of Functional Programs — A Bibliography.
Functional Language Implementation Project Document 18, Department of Computer Science, University of Warwick, April 1990.

Abstract: A bibliography of documents relating to the execution of functional programs in a parallel environment is presented together with abstracts of (some of) the documents.

Keywords: Bibliography.

[Karlson, 1980]
K. Karlson.
Nebula — A Functional Operating System.
Technical report, Laboratory for Programming Methodology, Department of Computer Science, Chalmers University, Sweden, 1980
Keywords: Operating System.

[Kasai and others, 1980]
Kasai et al.
An Automatic Time Analysis System.
Technical Report RIMS-335, Kyoto University, Japan, May 1980.

Abstract: This paper describes a system, TIMER, that generates upper and lower bounds for the complexity of programs written in a small imperative C-like language. The system is semi-automatic in that it requires some user annotation, and the estimates can be improved by further annotation of the program. As such it is a powerful tool for developing efficient algorithms, but it is not intended to be a program analysis system.

Keywords: Granularity Analysis.

[Katz and Weise, 1988]
Morry Katz and Daniel Weise.
Continuing Into the Future: On the Interaction of Futures and First-Class Continuations (A Capsule Summary).
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 101–102, Sendai, Japan, June 5–8, 1988. Springer, Berlin, Full paper published in 1990 ACM Conference on Lisp and Functional Programming.

Abstract: One of the nicest features of the *future* construct originally presented in Multilisp is its near orthogonality with respect to a functional subset of Scheme. Introducing futures into most functional programs does not affect the value returned, even though the parallel execution order might differ from the sequential. When futures and continuations are used in the same program, however, parallel and sequential executions can yield different results. No existing implementation of futures has yet addressed this issue. We make futures and continuations interact properly through a simple, yet important, change to the implementation of the future construct. This change causes a second problem to manifest itself: the creation of extraneous computation threads. The second problem is addressed by making an additional change to the future construct.

Keywords: Lisp, Futures, Continuations.

[Keller and Lin, 1984]
R. M. Keller and F. C. H. Lin.
Simulated Performance of a Reduction-Based Multiprocesor.
IEEE Computer, 17(7):70–82, July 1984
Keywords: Simulation, Graph Reduction.

[Keller, 1980]
Robert M. Keller.
Divide and CONCer: Data Structuring in Applicative Multiprocessing Systems.
In *1980 LISP Conference*, pages 196–202, Stanford, California, August 25–27, 1980.

Abstract: The conc representation of lists in implementations of applicative languages is introduced. This representation is based on an operator conc which supplements the usual cons, the representation itself being able to coexist with the conventional nested-pair representation. Advantages of the conc representation are presented which relate to multiprocessing and conventional systems. Qualitative comparison with other compact list representations is made. The conc representation exploits demand-driven evaluation and provides further justification for the "tagged-data" approach.

Keywords: Lisp, Distributed Data Structures.

[Keller, 1985]
R. M. Keller.
Distributed Computation by Graph Reduction.
Systems Research, 2(4):285–295, 1985.

Abstract: The author describes the use and distributed implementation of a functional language based on the graph reduction concept. Attention is given to the issue of implementing higher-order functions and concurrent evaluation. Prospects for optimization of this implementation are pointed out.

Keywords: Parallel Graph Reduction.

[Kelly, 1989]
Paul Kelly.
Functional Programming for Loosely-coupled Multiprocessors.
Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

Abstract: The author's Ph.D. thesis gives an extensive overview on the state of the art in (parallel) functional programming and describes several techniques for the construction of parallel functional programs. The para-functional programming language Caliban is introduced that allows the declarative description of processor networks and the mapping of processes to processors.

Keywords: Para-Functional Languages, Process Mapping, Streams.

[Kennaway and Sleep, 1982a]
J. R. Kennaway and M. R. Sleep.
Expressions as Processes.
In *1982 Symposium on Lisp and Functional Programming*, pages 21–28. ACM Press, New York, 1982
Keywords: Lambda-Calculus, Processes.

[Kennaway and Sleep, 1982b]
J. R. Kennaway and M. R. Sleep.
Parallel Implementations of Functional Languages.
In *1982 International Conference on Parallel Processing*, pages 168–170. IEEE, August 1982
Keywords: Parallel Functional Programming.

[Kesseler, 1991]
Marco Kesseler.
Implementing the PABC Machine on Transputers.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 409–421, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: In this paper an efficient implementation of the PABC machine on transputers is presented. The PABC machine is an abstract machine that is used to implement Concurrent Clean, an experimental lazy functional language. For a single transputer this means an 'unbounded' graph store has to be implemented and a set of reducers, each having a couple of stacks. In a network of transputers each reducer should be able to send graphs to other transputers, start new reducers on these remote graphs and to get reduced graphs back. At the moment the PABC machine runs on a single transputer only. Efficient and reliable implementation on a network of transputers has started.

Keywords: Term Graph Reduction, Abstract Machine.

[Kessler and Swanson, 1988]
Robert R. Kessler and Mark R. Swanson.
Concurrent Scheme.
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 200–234, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

Abstract: This paper describes an evolution of the Scheme language to support parallelism with tight coupling of control and data. Mechanisms are presented to address the difficult and related problem of mutual exclusion and data sharing which arise in concurrent language systems. The mechanisms are tailored to preserve Scheme semantics as much as possible while allowing for efficient implementation. Prototype implementations of the resulting language are described which have been completed. A third implementation is underway for the Mayfly, a distributed memory, twisted-torus communication topology, parallel processor, under development at the Hewlett-Packard Research Laboratories. The language model is particularly well-suited for the Mayfly processor, as will be shown.

Keywords: Lisp, Threads.

[Kewley and Glynn, 1989]
John M. Kewley and Kevin Glynn.
Evaluation Annotations for Hope+.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 329–337, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

Abstract: To fully exploit the parallelism of the Flagship Machine it is desirable to be able to experiment with a variety of evaluation strategies. The ability to control the behavior of individual functions via annotations is of enormous help in understanding the behavior of a parallel system and permits the comparison of "lazy" and "eager" versions of languages as well as providing the opportunity to intermix the two evaluation orders. The primary functional language used by the Flagship project is Hope+. Hope+ has a mixed evaluation strategy, with strict function application combined with lazy data constructors. This can result in both unnecessary evaluation and loss of parallelism. Annotations enable user-control of the evaluation strategy so that such problems are avoided.

Keywords: Para-Functional Programming, Annotations.

[Kieburtz, 1986]
Richard B. Kieburtz.
When Chasing your Tail Saves Time.
Information Processing Letters, 23(6):321–324, December 1986.

Abstract: Several investigators have observed experimentally that, in evaluating functional language by graph reduction, the time required for evaluation depends rather strongly on whether or not cyclic data structures are used to represent recursively defined expressions. The result is surprising, and the purpose of this short article is to explain it.

Keywords: Circular Programs.

[King and Wadler, 1992]
David J. King and Philip Wadler.
Combining Monads.
In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow*, Workshops in Computing, Ayr, Scotland, 1992. Springer, Berlin.

Abstract: Monads provide a way of structuring functional programs. Most real applications require a combination of primitive monads. Here we describe how some monads may be combined with others to yield a *combined monad*.

Keywords: Monads, State.

[Kluge, 1983]
W. E. Kluge.
Cooperating Reduction Machines.
Transactions on Computers, C 32(11):1002–1012, 1983
Keywords: Graph Reduction.

[Kluge, 1992]
Werner Kluge.
The Organization of Reduction, Data Flow, and Control Flow Systems.
MIT Press, 1992
Keywords: Graph Reduction, Data Flow.

[Koorey Willcock, 1992]
Diane M. Koorey Willcock.
Tlisp: A Concurrent Lisp for the Transputer.
SIGSAM Bulletin, 26(4):15–23, November 1992.

Abstract: Tlisp is an experimental concurrent lisp based on Xlisp. It is designed to run on an array of transputers with the Helios operating system. The eval-server model of parallelism is used with each transputer in the network reading, evaluating and printing s-expressions independently of the others. Tlisp has been developed for work with computer algebra algorithms, in particular, an interpolation algorithm for computing greatest common divisors of multivariate polynomials. Satisfactory performance has been observed with this application.

Keywords: Lisp.

[Kranz *et al.*, 1988]
David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr.
Mul-T: A High-Performance Parallel Lisp (Extended Abstract).
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 306–311, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

Abstract: Mul-T is a parallel Lisp system that has been developed to run on an Encore Multimax processor. Mul-T is an extended version of the T version 3 (or "T3") system that supports parallel processing using Multilisp's `future` construct. Multilisp's implementation uses a layer of interpretation that limits its speed, but Mul-T uses T3's ORBIT compiler (suitably modified) to generate native code for the Multimax's NS32332 processors, leading to a dramatic increase in speed (about a factor of 100 over the Multilisp system).

Keywords: Lisp, Futures.

[Kranz *et al.*, 1989]
David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr.
Mul-T: A High-Performance Parallel Lisp.
In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, number 24(7) in SIGPLAN Notices, pages 81–90, Portland, Oregon, June 21–23, July 1989. ACM Press, New York.

Abstract: Mul-T is a parallel Lisp system, based on Multilisp's `future` construct, that has been developed to run on an Encore Multimax processor. Mul-T is an extended version of the Yale T system and uses the T system's ORBIT compiler to achieve "production quality" performance on stock hardware — about 100 times faster than Multilisp. Mul-T shows that futures can be implemented cheaply enough to be useful in a production-quality system. Mul-T is fully operational, including a user interface that supports managing groups of parallel tasks.

Keywords: Lisp, Futures.

[Kubiak *et al.*, 1991]
R. Kubiak, J. Hughes, , and J. Launchbury.
A Prototype Implementation of Projection-Based First-Order Polymorphic Strictness Analysis.
In Philip Wadler, editor, *Functional Programming: Proceedings of the 1991 Glasgow Workshop*, Springer Workshops in Computing, Isle of Skye, Scotland, August 13-15, 1991. Springer, Berlin.

Abstract: This paper discusses the implementation of first-order, polymorphic Backwards Strictness Analyzer in the Glasgow Haskell Compiler.

Keywords: Strictness Analysis, Projections.

[Kuchen and Gladitz, 1992]
Herbert Kuchen and Katia Gladitz.
Implementing Bags on a Shared Memory MIMD-Machine.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: Multisets (also called bags) are an interesting data structure for parallely implemented functional programming languages, since they do not force an unneeded restriction of the data flow and allow to exploit as much parallelism as possible. Most operations on multisets can be understood as special cases of the so-called Gamma scheme. In the present paper, we investigate efficient implementations of several instances of this Gamma scheme on MIMD-machines with shared memory.

Keywords: Multisets, Bags, Non-Determinism.

[Kuchen and Loogen, 1989]
Herbert Kuchen and Rita Loogen.
Parallele Implementierung einer funktionalen Programmiersprache auf einem Transputer-Mehrprozessor-System (Parallel Implementation of a Functional Programming Language on a Multitransputer-System) (in German).
In *1. Transputer-Anwender-Treffen*, Informatik-Fachberichte. Springer, Berlin, 1989.

Abstract: In this paper we describe the parallel implementation of a functional programming language on a multitransputer system. First a functional program is translated into a system of special function definitions called serial combinators. Herewith the implicit parallelism to be utilized is made explicit. The system of serial combinators is transformed into code for a parallel abstract machine based on graph reduction. Furthermore some data structures of the functional language are introduced and compared. For this purpose, the runtimes and speedups measured for some example programs are given.

Keywords: Parallel Graph Reduction, Serial Combinators, Evaluation Transformers, Distributed Data Structures.

[Kuchen, 1989]
Herbert Kuchen.
Applikative Datenstrukturen in der parallelen abstrakten Maschine PAM (Applicative Data Structures in the Parallel Abstract Machine PAM) (in German).
In Walter Dosch, editor, *Funktionale und Logische Programmierung — Sprachen, Methoden, Implementationen*, Report 214, University Augsburg, Germany, December 1989
Keywords: Distributed Data Structures.

[Kurfeß *et al.*, 1990]
F. Kurfeß, X. Pandolfi, Z. Belmesk, W. Ertel, R. Letz, and J. Schumann.
PARTHEO and FP2: Design of a Parallel Inference Machine.
In P. C. Treleaven, editor, *Parallel Computers — Object-Oriented, Functional, Logic*, Series in Parallel Computing, chapter 9, pages 259–297, John Wiley & Sons, Chichester, 1990.

Abstract: The language FP2 provides a simple and consistent set of constructs for the description of parallel systems. Such a parallel system is seen as a network of concurrent communicating processes which, apart from their specified interconnections, are independent. Computations to be performed are expressed in the form of a functional specification of the behavior of processes via rewrite rules; communication consists of the exchange of information between processes trough ports with an extended rendezvous mechanism based on unification.

Keywords: FP2, Process Networks.

[Langendoen, 1991]
Koen Langendoen.
WYBERT: Parallel Reduction on Shared Memory.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 219–223, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: The Dutch Parallel Reduction Machine project has shown that it is possible to implement parallel reduction based on a coarse-grain model. Within the framework of this project a prototype machine has been constructed: the Amsterdam Parallel Experimental Reduction Machine (APERM). Experimental results indicate that the architecture can be improved by connecting several processors to one graph memory. The design of APERM2 is based on this idea and consists of an extensible number of processor clusters (shared-memory multiprocessors) interconnected by a communication network. The distribution of parallel tasks over the clusters will be done by the same coarse-grain reduction model as adopted in APERM (divide&conquer), while parallel reduction inside a cluster is the research topic of WYBERT. The objective of WYBERT is to study the opportunities to exploit shared memory, and its associated caches, for parallel graph reduction of divide&conquer applications.

Keywords: Graph Reduction, Shared Memory, Runtime System.

[Larus and Hilfinger, 1988]
James R. Larus and Paul N. Hilfinger.
Restructuring Lisp Programs for Concurrent Execution.
In *PPEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems*, volume 23(9) of *SIGPLAN Notices*, pages 100–110, New Have, Connecticut, July 19–21, 1988. ACM Press, New York.

Abstract: This paper describes the technique that the program transformation system CURARE uses to restructure Lisp programs for concurrent execution in Multiprocessor Lisp systems and discusses the problems inherent in producing concurrent programs in a flexible and dynamic programming language such as Lisp. CURARE's overall organization is similar to other program restructuring systems: it detects potential conflicts between statements in a program, then transforms the program to improve its concurrent performance, and finally inserts synchronization to ensure the program's concurrent behavior. However, the language and programs that CURARE transforms are very different from the FORTRAN programs that are the traditional targets of program restructuring and so CURARE requires new algorithms and approaches, which are described in this paper.

Keywords: Lisp, Parallelization.

[Le Metayer, 1985]
Daniel Le Metayer.
Mechanical Analysis of Program Complexity.
In *SIGPLAN '85 Symposium*, volume 20(7) of *SIGPLAN Notices*, pages 69–73. ACM Press, New York, 1985.

Abstract: This paper describes the ACE system for automatic generation of complexity functions for a subset of FP. This system is based on program transformation, with a library of more than 1000 transformation rules which are applied to transform the step-counting version of a program to a composition of the complexity function and a size measure.

Keywords: Granularity Analysis.

[Le Metayer, 1988]
Daniel Le Metayer.
ACE: An Automatic Complexity Evaluator.
ACM Transactions on Programming Languages and Systems, 10(2):248–266, April 1988.

Abstract: There has been a great deal of research done on the evaluation of the complexity of particular algorithms; little effort, however, has been devoted to the mechanization of this evaluation. The ACE (Automatic Complexity Evaluator) system is able to analyze reasonably large programs, like sorting programs, in a fully mechanically way. A time-complexity function is derived from the initial functional program. This function is transformed into its nonrecursive equivalent according to McCarthy's recursion induction principle, using a predefined library of recursive definitions. As the complexity is not a decidable property, this transformation will not be possible in all cases. The richer the predefined library is, the more likely the system is to succeed. The operations performed by ACE are described and the use ofthe system is illustrated with the analysis of a sorting algorithm. Related works and further improvements are discussed in the conclusion.

Keywords: Granularity Analysis.

[Lee *et al.*, 1988]
Ching-Cheng Lee, Stephen Skedzielewski, and John Feo.
On the Implementation of Applicative Languages on Shared-Memory, MIMD Multiprocessors.
In *PPEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems*, number 23(9) in SIGPLAN Notices, pages 188–197, New Haven, Conneticut, July 19–21, September 1988. ACM Press, New York.

Abstract: This paper presents the performance of a set of algorithms written in Sisal and run on multiprocessor Sequent, DEC, and Cray computers. We describe our current runtime system and discuss its implementation on each machine. We indicate where our automatic approach to parallelization works well, as well as sources of inefficient behavior. Overall we find our systems encouraging for the first release of the native-code generating software. We suggest improvements to the compiler and runtime systems, many already developed but not yet implemented, to alleviate the inefficiencies.

Keywords: SISAL, Algorithms.

[Lee, 1988]
M. K. O. Lee.
A Parallel Computational Model for Functional Languages and Its Formal Specification in CSP.
In *CONPAR 88*, volume C, pages 9–16, Manchester, UK, September 12–16, 1988. British Computer Society.

Abstract: This paper presents a parallel and distributed computational model for functional languages based on the notion of demand/data driven reduction of graphical combinators. A graphical combinator language is used as the canonical language form for the computational model. The behavior of the model is formally described using Hoare's CSP constructive specification of a detailed and practical parallel computational model of this kind. CSP is found to be a suitable and valuable tool for conveying complex design ideas within a parallel and distributed framework.

Keywords: Semantics, CSP.

[Lemaitre *et al.*, 1986]
M. Lemaitre, M. Castan, M.-H. Durand, G. Durrieu, and B. Lecussan.
Mechanisms for Efficient Multiprocessor Combinator Reduction.
In *1986 ACM Conference on Lisp and Functional Programming*, pages 113–121, Cambridge, Massachusetts, August 4–6, 1986. ACM Press, New York.

Abstract: We are currently designing a multiprocessor architecture for a parallel reduction machine named MaRS, based on a combinator machine language. Since the seminal papers, several projects and studies close to our own have appeared, such as the SKIM2 processor, the DAPS model, the COBWEB project. Other studies like the rediflow multiprocessor or the Multilisp experiments, although involving different hypotheses, are worth considering in this context. So, this paper will just describe briefly the MaRS machine, and focus on more original features which, as far as we know, do not appear in other projects.

Keywords: Parallel Graph Reduction, Combinators.

[Lester and Burn, 1989]
D. R. Lester and G. L. Burn.
An Executable Specification of the HDG-Machine.
Technical report, GEC Hirst Research Centre, November 1989
Keywords: Parallel Graph Reduction.

[Lester, 1989]
David R. Lester.
Stacklessness: Compiling Recursion for a Distributed Architecture.
In *FPCA '89, Functional Programming Languages and Computer Architecture*, pages 116–128, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: Compiling general programming languages to run efficiently on a distributed architecture is hard. One of the problems that confronts the potential implementor, is how to store the stack. This is normally used in two ways: firstly it is used to hold the arguments and temporary variables of a function or procedure, and secondly it is used to hold a previous state on entry to a new function or procedure. One way to represent such a stack is as a contiguous array. An alternative is to hold the arguments and temporary variables in a stack frame, with a back-link to the previous stack frame. In a parallel machine each task will require a separate stack. We implement each of these stacks as a linked list of stack frames, each of which resides in a garbage collected heap. Using the stacklessness analysis, a node which requires evaluation can be created with a stack frame large enough to evaluate all tail recursive calls that may occur in the reduction sequence. It is therefore unnecessary to provide an extension mechanism which enlarges stack frames. The stacklessness analysis involves giving a non-standard semantics to a typed functional language. The technique may be applied to any resource with a stack-like pattern of consumption. The paper includes a proof that an approximation to the fixpoint of a combinator in the abstract interpretation is computable. The LaTeX source of the paper is also an executable Orwell program to find this approximation. The problem is particularly difficult when higher order functions are used. The stacklessness analysis is able to deal with this case. The analysis is used in the HDG-machine, a compiler and abstract machine for a Transputer based implementation of distributed graph reduction.

Keywords: Parallel Graph Reduction, Compilation, Tail-Recursion.

[Li and Hudak, 1986]
Kai Li and Paul Hudak.
A New List Compaction Method.
Software — Practice and Experience, 16(2):145–163, February 1986.

Abstract: List compaction, or so-called 'cdr-coding', can greatly reduce the storage needs of list processing languages. However, existing methods do not perform well when several lists are being constructed simultaneously from the same heap, since the non-contiguous nature of the cells being allocated eliminates the opportunity for compaction. This situation arises not only in true parallel systems sharing a common memory, and sequential systems supporting multiple processes, but also quite often in purely sequential systems, where it is not uncommon to build several different lists simultaneously within a single loop. In this paper, a new list compaction method is presented that performs well during both sequential and 'parallel' list generation. The method is essentially a generalization of cdr-coding, in which lists are represented explicitly as linked vectors rather than implicitly as compacted memory. In addition, an encoding scheme is used that is simple or simpler than all known encodings, and destructive operations are supported with no greater overhead than existing schemes. Performance figures in a simulated environment suggest that the strategy consistently performs better than conventional cdr-coding, with essential the same complexity.

Keywords: List compaction.

[Livezey and Muntz, 1989]
Brian K. Livezey and Richard R. Muntz.
ASPEN: A Stream Processing Environment.
In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, volume 365 of *Lecture Notes in Computer Science*, pages 374–388, Eindhoven, The Netherlands, June 12–16, 1989. Springer, Berlin.

Abstract: In this paper, we describe ASPEN, a concurrent stream processing environment. ASPEN is novel in that it provides a programming model in which programmers use simple annotations to exploit varying degrees and types of concurrency. The degree of concurrency to be exploited is not fixed by the program specification or by the underlying system. Increasing or decreasing the degree of concurrency to be exploited during execution does not require rewriting the entire program, but rather, simply re-annotating it. Examples are given to illustrate the varying types of concurrency inherent in programs written within the stream processing paradigm. We show how programs may be annotated to exploit these varying degrees of concurrency. We briefly describe our implementation of ASPEN.

Keywords: Para-Functional Programming, Annotations, Streams.

[Loidl, 1992a]
Hans Wolfgang Loidl.
A Parallelizing Compiler for the Functional Programming Language EVE.
Master's thesis, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992, Also: Technical Report 92-30, RISC-Linz, Johannes Kepler University, Linz, Austria.

Abstract: This thesis describes the design and implementation of an automatically parallelizing compiler for the purely functional programming language EVE. The target language of the compilation process is the assembler code of the abstract dataflow machine ADAM that has been implemented in the frame of another thesis. In this compilation process dataflow graphs are used as an intermediate language. These dataflow graphs must then be translated into a controlflow program with explicit statements for creating and synchronizing parallel threads of control. A prototype version of the EVE compiler is running.

Keywords: Compilation, Dataflow, Controlflow.

[Loidl, 1992b]
Hans Wolfgang Loidl.
A Parallelizing Compiler for the Functional Programming Language EVE (Extended Abstract).
In Szabolcs Ferenczi, editor, *Proceedings of the 1st Austrian-Hungarian Workshop on Transputer Applications*, number KFKI-1992-34/M,N in Technical Report Series of the Hungarian Academy of Sciences Central Research Insitute for Physics, pages 1–10, Sopron, Hungary, October 8–10, 1992.

Abstract: This paper describes the compilation process from the non-strict functional programming language EVE to the assembler code of the abstract hybrid dataflow/controlflow machine ADAM. In this compilation process dataflow graphs are used as an intermediate language. However, the resulting assembler code is a controlflow program that simulates the behavior of the dataflow graph. The non-strict semantics of EVE ensures that it contains maximal inherent parallelism. The target language of the compilation process is ADAM assembler code which is essentially a controlflow code that is augmented with instructions for explicit creation and synchronization of parallel tasks. The compiler is written in Modula-2 and runs on a PC.

Keywords: Compilation, Dataflow, Controlflow.

[Loidl, 1992c]
Hans Wolfgang Loidl.
Circular Programs on Compound Data Structures.
Technical Report 92-06, RISC-Linz, Johannes Kepler University, Linz, Austria, January 1992.

Abstract: A circular program creates a data structure whose computation depends upon (parts of) the structure itself. This paper provides a survey of what has been published about circular programs during the last years. So, in this paper it is shown that a non-strict semantics and local recursion are necessary properties of the underlying functional language. Furthermore, it will be shown that circular programs can have a better efficiency (e.g. by avoiding the production of intermediate temporary structures or by being able to traverse a compound data structure only once) than conventional programs. As the termination of a circular program is by no means obvious, a strategy will be outlined how to prove if every element of a list that is defined by a circular program can be computed in finite time.

Keywords: Circular Programs, Streams.

[Loogen *et al.*, 1989]
Rita Loogen, Herbert Kuchen, Klaus Indermark, and Werner Damm.
Distributed Implementation of Programmed Graph Reduction.
In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, volume 365 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 12–16, 1989. Springer, Berlin.

Abstract: Programmed graph reduction has been shown to be an efficient implementation technique for lazy functional languages on sequential machines. Considering programmed graph reduction as a generalization of conventional environment-based implementations where the activation records are allocated in a graph instead of on a stack it becomes very easy to use this technique for the execution of functional programs in a parallel machine in distributed memory. We describe in this paper the realization of programmed graph reduction in PAM — a parallel abstract machine with distributed memory. Results of our implementation of PAM on an OCCAM-Transputersystem are given.

Keywords: Programmed Graph Reduction, Serial Combinators, Evaluation Transformers.

[Loogen, 1987]
Rita Loogen.
Design of a Parallel Programmable Graph Reduction Machine With Distributed Memory.
Aachener Informatik-Berichte 87-11, RWTH Aachen, Fachgruppe Informatik, Aachen, Germany, 1987.

Abstract: This paper describes the systematic design of a parallel programmable graph reduction machine with distributed memory. The abstract architecture that is presented is not just one of many proposals in this area, but highlights the essential features of such machines. Parallel program execution will naturally cause additional organizational overhead (message handling, task distribution, workload balancing) that will certainly slow down the reduction process if only one processor is available in each node of the multicomputer system. To avoid such loss in efficiency each processor node of our abstract architecture consists of two processing units — the reduction unit and the input/output unit — which work in parallel exchanging messages in a shared local memory. The reduction unit performs the program evaluation by programmed graph reduction, while the input/output unit keeps organizational work that is important for the parallel program execution, but can be done independently from reduction process, away from the reduction unit. In addition to the gain in efficiency this organization of the abstract architecture allows to discuss the issues of the graph reduction process, the parallelization of this process and the structure of the communication network separately.

Keywords: Programmed Graph Reduction.

[Loogen, 1990]
Rita Loogen.
Parallele Implementierung Funktionaler Programmiersprachen (Parallel Implementation of Functional Programming Languages) (in German).
Number 232 in Informatik Fachberichte, Springer, Berlin, 1990.

Abstract: The author's Ph.D. thesis describes the implementation of a lazy functional language on a transputer network by parallel graph reduction. The book contains an extensive section on the formal framework of compilation e.g. on lambda-lifting and on strictness analysis by evaluation transformers.

Keywords: Strictness Analysis, Evaluation Transformers, Parallel Graph Reduction.

[Maaßen, 1992]
Andreas Maaßen.
Parallel Programming with Data Structures and Higher Order Functions.
Science of Computer Programming, 18:1–38, 1992.

Abstract: Linear lists, which are the standard data structure in functional programming languages, have proved to be useful for many applications. Yet for several reasons it seems that there is a need for additional applicative data structures: Firstly, functional programming languages are well suited for parallel execution, because programs written in these languages cannot have side effects, and expressions can be evaluated independently. But parallel execution of functional programs is often hampered by the linear structure of lists, which supports only one element at a time algorithms. Secondly, lists lack the expressive power that we need in general purpose languages. For example, there is a need for array-like data structures. Thirdly, many functions on lists have to be defined recursively. No matter how efficiently recursion may be implemented, it will always fall short of the efficiency that can be obtained with imperative implementations. We propose a choice of data structures and corresponding higher order functions that enable programmers to write concise and (almost) recursion-free functional programs which have a higher degree of potential parallelism. Possible implementation techniques are discussed, and the major complexity result is that, under parallel evaluation, many functions on our data structures need only logarithmic time, whereas using lists results in linear time complexity.

Keywords: Distributed Data Structures.

[Maranget, 1991]
Luc Maranget.
GAML: a Parallel Implementation of Lazy ML.
In John Hughes, editor, *FPCA '91, Functional Programming Languages and Computer Architectures*, volume 523 of *Lecture Notes in Computer Science*, pages 102–123, Harvard, Massachusetts, USA, 1991. Springer, Berlin.

Abstract: We present a new parallel implementation of Lazy ML. Our scheme is a direct extension of the G-machine-based implementation of Lazy ML. Parallelism is introduced by *fork* annotations inserted by the programmer. We discuss the interference of such user annotations with strictness annotations generated by our compiler. The system has been implemented on a Sequent Balance computer. We also address the main practical issues involved, including stack and heap management.

Keywords: Parallel Graph Reductoin, G-Machine, Annotations.

[Marino and Succi, 1989]
Giuseppe Marino and Giancarlo Succi.
Data Structures for Parallel Execution of Functional Languages.
In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, volume 365 of *Lecture Notes in Computer Science*, pages 346–356, Eindhoven, The Netherlands, June 12–16, 1989. Springer, Berlin.

Abstract: After a short review of some data structures proposed by traditional functional languages, some requirements that a data structuring primitive should meet for suitable implementation on parallel architectures are introduced; the drawbacks of the classic *list* data structure are evidenced and consequently two alternative structures are proposed; the first is based on unordered collection of objects (and hereafter called *bag*); its formal definition is given and the non-determinism of the related primitive operators are discussed; the second, *limited function*, is a way of defining functions over finite and countable domains; its significant implications, mainly related with parallel architectures, are described. Examples of programs based on these concepts are integrated within the framework of an existing functional language are proposed and properties about the primitives for manipulating unordered collections of objects are stated and proved in Appendix.

Keywords: Distributed Data Structures, Non-Determinism.

[Marti, 1980]
Jed B. Marti.
Compilation Techniques for a Control-Flow Concurrent Lisp System.
In *1980 LISP Conference*, pages 203–207, Stanford, California, August 25–27, 1980.

Abstract: Described is a strategy for the compilation of LISP programs for a concurrent processing system in the presence of global side effects. The method supports "horizontal" concurrency, the concurrency available when two arguments of a function can be evaluated simultaneously and without violating the semantics of sequential LISP. The method works for Standard LISP and is not restricted to "pure" LISP. Conventional programs may usually be compiled without modification.

Keywords: Lisp, Parallelization, Compilation.

[Martin and Hankin, 1987]
Chris Martin and Chris Hankin.
Finding Fixed Points in Finite Lattices.
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 426–445, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

Abstract: Recently there has been much interest in the abstract interpretation of declarative languages. Abstract interpretation is a semantics-based approach to program analysis that uses compile time evaluation of programs using simplified value domains. This gives information about the run-time properties of programs and provides the basis for significant performance improvements. A particular example of abstract interpretation is strictness analysis which allows the detection of parameters in which a function is strict; these parameters may be passed by value without compromising the termination properties of the program. The central, most complex task of an abstract interpreter is finding the fixpoints of recursive functions in the abstract value space. An elegant algorithm, the frontiers algorithm, has been proposed by Simon Peyton-Jones and Chris Clack that performs very well for the strictness analysis of first-order functions. In this paper we extend their algorithm and show how it can be applied to higher-order functions over arbitrary fine lattices. This raises the possibility of using the algorithm as the basis for more general abstract interpretation tools. We describe the algorithm in a modular way that is conducive to proofs of correctness and termination properties.

Keywords: Abstract Interpretation, Fixed Points, Frontiers Algorithm.

[Martin, 1989]
Chris Martin.
Algorithms for Finding Fixpoints in Abstract Interpretation.
PhD thesis, Department of Computing, Imperial College, London, UK, 1989
Keywords: Abtract Interpretation, Fixpoints.

[Martins, 1992]
Wellington Santos Martins.
Parallel Implementations of Functional Languages.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: With interest in parallel implementations of functional languages as a whole, we first discuss some issues related to parallel execution of functional programs, and then review some key proposals considering three different approaches, namely Parallel Lisp Systems, Dataflow Systems and Reduction Systems

Keywords: Parallel Lisp, Dataflow, Parallel Graph Reduction.

[Matthews, 1991]
David Matthews.
A Distributed Concurrent Implementation of Standard ML.
In *EurOpen Autumn Conference*, 1991
Keywords: ML, Threads.

[Maurer, 1985]
Dieter Maurer.
Strictness Computation Using Special λ-Expressions.
In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects, Proceedings of a Workshop*, volume 217 of *Lecture Notes in Computer Science*, pages 136–155, Copenhagen, Denmark, October 17–19, 1985. Springer, Berlin.

Abstract: In order to exploit the implicit parallelism in demand driven functional programs the inference of information about the strictness of user defined functions is helpful. This article extends results of Mycroft about strictness detection in first order programs to higher order programs. Information about the strictness of higher order functions is represented by special λ-expressions, so called need expressions. A program over λ-expressions is translated into a program over need expressions by interpreting the constants as closed need expressions. After a suitable approximation — essentially a partial typing — expressions are computed by an iterative method allowing to derive strictness information for the functions defined in the original program.

Keywords: Strictness Analysis.

[McCarthy, 1963]
John McCarthy.
A Basis for a Mathematical Theory of Computation.
In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963

Keywords: Non-Determinism.

[McGraw *et al.*, 1985]
James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas.
SISAL — Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2.
Technical Report M-146, Lawrence Livermore National Laboratory, University of California, March 1985.

Abstract: SISAL (Streams and Iteration in a Single-Assignment Language) is a functional data-flow language intended for use on a variety of sequential, vector, multi-, and data-flow processors. SISAL is designed to express algorithms fore execution on computers capable of highly concurrent operation. More specifically, the application area to be supported is numerical computation which strains the limits of high performance machines, and the primary targets for translation of SISAL programs are dataflow data-driven machines. Nevertheless, it has been our intention that the language not have idiosyncrasies reflecting the particular nature of the application area or target machine. It should be reasonable for SISAL to evolve into a general purpose language appropriate for writing programs to run on future general parallel computers.

Keywords: SISAL, Dataflow.

[McKay and Shapiro, 1980]
Donald P. McKay and Stuart C. Shapiro.
MULTI — A LISP Based Multiprocessing System.
In *1980 LISP Conference*, pages 29–37, Stanford, California, August 25–27, 1980.

Abstract: A package of LISP functions, collectively called MULTI, which extends LISP 1.5 to multiprogramming is presented. MULTI defines the notion of a process within a LISP implementation using function invocation as the only control primitive. A process is an executable entity consisting of a process template and a set of register values. The process template defines the operations the process carries out. Process environments are saved in what can be viewed as function call instances, i.e. LISP forms which have the name of a process template in functional position and the register values following it. The flexibility of this simple conceptualization of processes is demonstrated by several examples which use MULTI to implement recursion, backtracking, generators, agendas and AND/OR graph searching. The implementation of MULTI does not assume that the host LISP system provides any data or control environment saving mechanisms such as funarg or Interlisp's spaghetti stack. Thus, MULTI is portable to other LISP systems.

Keywords: LISP.

[Merrall, 1991a]
Simon Merrall.
Plurals — A SIMD extension to Eulisp.
Technical report, School of Mathematical Sciences, University of Bath, 1991.

Abstract: We describe in this report a massively parallel SIMD extension to EuLisp our local parallel dialect of Lisp, called Plurals. The extensions allow the programmers to allocate Lisp objects on a data parallel processor array, in our case a MasPar MP-10111. The physical size of the processor array is hidden from the programmer who can request different sized sets of virtual processing elements. This gives a mechanism for working with datasets with fewer elements than the processor array whilst leaving the remaining PEs free for other datasets. This has removed us one level from the traditional global execution model of data parallel processing.

Keywords: Lisp, SIMD.

[Merrall, 1991b]
Simon Merrall.
TPL — An Implementation of Paralation Lisp in EuLisp.
Technical report, School of Mathematical Sciences, University of Bath, 1991.

Abstract: The department has recently acquired a Maspar MP-1, a massively parallel SIMD computer which has 1024 processors scalable to 16384 processors. We have been looking for an appropriate model of SIMD computation to allow us to use the MasPar with EuLisp alongside our existing MIMD models. In this paper we describe the Paralation Model and an experimental serial implementation of Paralation Lisp written in EuLisp.

Keywords: Lisp, SIMD.

[Michaelson *et al.*, 1993]
Greg Michaelson, Tore Bratvold, David Busvine, and Kevin Waugh.
Parallel Implementations from Functional Prototype Instrumentation.
In *PARCO' 93*, Grenoble, France, September, 1993. Elsevier North-Holland.

Abstract: We use the dynamic instrumentation of prototypes built from higher-order functions (HOFs) to decide where parallelism is best exploited. Performance models relate HOFs behaviour in prototypes to parallel harness behaviour in implementations. Where useful parallelism is identified, generic imperative parallel harnesses are used to implement HOFs. Otherwise, standard functional to sequential imperative translation is employed. Our prototypes are built in a pure functional subset of Standard ML. Final implementations are in Occam2 on a transputer based Meiko Computing Surface.

Keywords: Algorithmic Skeletons, Granularity Control, ML.

[Michelson *et al.*, 1986]
R. Michelson, L. Smith, E. Williams, and B. Yantis.
Parallel Graph Reduction on a Supercomputer: A Status Report.
In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction: Proceedings of a Workshop*, volume 279 of *Lecture Notes in Computer Science*, pages 114–118, Santa Fe, New Mexico, September 29 – October 1, 1986. Springer, Berlin.

Abstract: We describe an ongoing effort to develop a parallel graph reduction run-time system hosted on a multiprocessor supercomputer. This run-time system is presented augmented by the functional language compiler of the ALFALA system. Admittedly, parallel graph reduction is hardly a novel idea. The interesting notion is the provision of a parallel execution environment sufficiently powerful to support development of large prototypical scientific and symbolic codes in a functional language. This will allow the investigation of the functional programming paradigm within these application domains in an empirical fashion. In this paper, we discuss the motivation for the effort, describe the basic elements of the implementation, and provide some preliminary insight distilled from our experience with an initial version of the runtime system.

Keywords: Parallel Graph Reduction, Runtime System.

[Milewski and Mori, 1990]
Jaroslaw Milewski and Imrich Mori.
Efficient Loop Handling in a Stream-Oriented Unraveling Dataflow Interpreter.
Computers and Artificial Intelligence, 9(6):545–570, 1990.

Abstract: We propose a detailed implementation of an unraveling dataflow interpreter of the Id programming language, based on table encoding of activation names. We concentrate on loop programming construct and on label-manipulating operators of the underlying dataflow model. Moreover, we show how to represent and process the stream, being the basic data structure in classical Id. Since streams are processed by loops, the algorithmic attractiveness of the former meets in our model the efficient implementation of the latter.

Keywords: Dataflow, Id, Streams.

[Milikowski and Vree, 1991]
R. Milikowski and W. G. Vree.
The G-Line — A Distributed Processor for Graph Reduction.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 265–282, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: The G-line is a horizontally coded graph reducer. In lazy functional languages much time is used to manage the graph. A method has been developed to perform construction of a subgraph in a single parallel access to the graph memory. With a simulation, using infinite hardware, we have shown that the architecture performs well. We argue that a realistic implementation is possible.

Keywords: Parallel Graph Reduction, Abstract Machine.

[Miller, 1987]
J. S. Miller.
MultiScheme, A Parallel Processing System Based on MIT Scheme.
PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1987, Also: Technical Report MIT LCS/TR/402
Keywords: Lisp, Parallelization.

[Miller, 1988]
J. S. Miller.
Implementing a Scheme-Based Parallel Processing System.
International Journal of Parallel Programming, 17(5):367–402, October 1988
Keywords: Lisp, Parallelization.

[Mintchev, 1992]
Sava Mintchev.
Using Strictness Information in the STG-Machine.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: The paper presents an attempt at exploiting strictness information for parallel evaluation of functional programs. A simple evaluation model, which uses strictness in a limited way, is suggested. It has been applied in a parallel version of the STG-machine; special attention has been paid to avoiding the creation of useless tasks. Some results from the simulation of the parallel STG-machine are provided, throwing light on the amount and granularity of parallelism.

Keywords: Strictness Analysis, Parallel Graph Reduction.

[Model, 1980]
Mitchell L. Model.
Multiprocessing via Intercommunicating Lisp Systems.
In *1980 LISP Conference*, pages 188–195, Stanford, California, August 25–27, 1980.

Abstract: Multiprocess environments comprising several intercommunicating LISP systems are straightforward to implement due to certain fundamental characteristics of the LISP language. Experiences with four methods of establishing the necessary communications linkages are described. The features of LISP which support experimentation with interprocess communication are identified. Two key characteristics of the language are important in this regard: 1. LISP programs can construct and interpret new code as they run; 2. Structures within LISP systems are accessible by name. The most flexible of the communication methods use only the ordinary LISP input and output functions, supplemented by a small amount of system-dependent code to create communication linkages that can be treated by LISP as file structures.

Keywords: Lisp.

[Mohr *et al.*, 1990]
E. Mohr, D. Kranz, and R. Halstead.
Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs.
In *ACM Conference on Lisp and Functional Programming*, Nice, France, June 27–29, 1990. ACM Press, New York
Keywords: Granularity.

[Moor, 1982]
Ian W. Moor.
An Applicative Compiler for a Parallel Machine.
In *ACM SIGPLAN Symposium on Compiler Construction*, number 17(6) in SIGPLAN Notices, pages 284–293, Boston, Massachusetts, June 1982. ACM Press, New York, Also: Research Report Doc 83/6, Department of Computing, Imperial College, London, March 1983.

Abstract: A compiler for the applicative language HOPE is described. The compiler is itself written in HOPE and generates a machine independent compiler target language, suitable for execution on the parallel reduction machine ALICE. The advantages of writing a compiler in a very high level applicative language ae discussed, and the use of program transformation and other techniques to turn the initial "runnable specification" into a more efficient (if less clear) program are outlined. Extensions to the HOPE language and the compiler which can exploit the parallelism and various execution mode of ALICE are described.

Keywords: Graph Reduction, ALICE, HOPE.

[Morris and Schwarz, 1980]
F. Lockwood Morris and Jerald S. Schwarz.
Computing Cyclic List Structures.
In *1980 LISP Conference*, pages 144–153, Stanford, California, August 25–27, 1980.

Abstract: It is argued that list structures containing cycles are useful and unobjectionable Lisp entities. If this is so, it is desirable to have a means of computing them less foreign to the equational-definition style characteristic of Lisp than are the list-structure-altering primitives *rplaca* and *rplacd*. A notion is developed of a reasonable system of mutually recursive equations, guaranteed to have a unique solution is list structures. The notion is given in termis of the computations invoked by the equations, without reference to the forms of expressions appearing in them. A variety of programming examples are presented, including a curious implementation of the Knuth-Morris-Pratt string matching algorithm. Two methods of implementing the recursive definition facility are discussed.

Keywords: Circular Programs, Streams.

[Morrisett and Tolmach, 1992]
J. Gregory Morrisett and Andrew Tolmach.
A Portable Multiprocessor Interface for Standard ML of New Jersey.
Technical Report CMU-CS-92-155, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1992.

Abstract: We have designed a portable interface between shared-memory multiprocessors and Standard ML of New Jersey. The interface is based on the conventional kernel thread model and provides facilities that can be used to implement user-level thread packages. The interface supports experimentation with different thread scheduling policies and synchronization constructs. It has been ported to three different multiprocessors and used to construct a general purpose, user-level thread package. In this paper, we discuss the interface and its implementation and performance, with emphasis on the Silicon Graphics 4D/380S multiprocessor.

Keywords: ML, Threads.

[Morrisett, 1991]
J.G̃regory Morrisett.
Running your Continuation Threads in Parallel.
In *Third International Workshop on Standard ML*, Pttsburgh, PA, September 1991
Keywords: ML, Continuations, Threads.

[Mycroft and Jones, 1985]
Alan Mycroft and Neil D. Jones.
A Relational Framework for Abstract Interpretation.
In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects, Proceedings of a Workshop*, volume 217 of *Lecture Notes in Computer Science*, pages 156–171, Copenhagen, Denmark, October 17–19, 1985. Springer, Berlin.

Abstract: Abstract interpretation is a very general framework for proving certain properties of programs. This is done by interpreting the symbols of the program, or the symbols of a denotational metalanguage translation, in two different ways (the standard interpretation and the abstract interpretation) and relating them. We set up a new framework for abstract interpretation based on relations (with the intent of inclusive or logical relations). This avoids problems with power domains and enables certain higher-order frameworks to be proved correct. As an example we show how the Hindley/Milner type system can be viewed as a special case of our system and is thus automatically correct.

Keywords: Abstract Interpretation.

[Mycroft, 1986]
Alan Mycroft.
Abstract Interpretation and Optimising Transformations for Applicative Programs.
PhD thesis, University of Edinburgh, Scotland, 1986
Keywords: Abstract Interpretation.

[Nielson, 1989]
Flemming Nielson.
The Typed λ-Calculus with First-Class Processes.
In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, volume 365 of *Lecture Notes in Computer Science*, pages 357–373, Eindhoven, The Netherlands, June 12–16, 1989. Springer, Berlin.

Abstract: We extend the typed λ-calculus with CCS- or CSP-like processes and allow these to be first-class citizens just as functions are first-class citizens in functional languages. The main novel feature of the language is the use of *types* to record the *communication possibilities* possessed by processes and in this we give up the *causality* between communications, i.e. the types do not model whether or not one communication may takeplace before another. In analogy with the semantics of the λ-calculus, and of CCS, we develop a structural operational semantics for the language. We then prove that the operational semantics preserves the types and we use this to give examples of 'errors' that cannot arise for well-typed programs.

Keywords: Lambda Calculus, Processes, Semantics.

[Nikhil, 1987a]
Rishiyur S. Nikhil.
Id Nouveau, Reference Manual, Part I: Syntax.
Technical report, Computation Structures Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1987.

Abstract: This document is Part I of two reference manuals for Id Noveau, the language used to program the MIT Tagged-Token Dataflow Machine. This part describes the syntax of the language. Id Noveau is a functional programming language augmented with a parallel data-structuring mechanism called *I-structures*.

Keywords: Dataflow, Id Noveau.

[Nikhil, 1987b]
Rishiyur S. Nikhil.
Id Nouveau, Reference Manual, Part II: Operational Semantics.
Technical report, Computation Structures Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1987.

Abstract: This document is Part I of two reference manuals for Id Noveau, the language used to program the MIT Tagged-Token Dataflow Machine. This part describes the operational semantics of the language. Id Noveau is a functional programming language augmented with a parallel data-structuring mechanism called *I-structures*.

Keywords: Dataflow, Id Noveau, Operational Semantics.

[Nikhil, 1987c]
Rishiyur S. Nikhil.
Id World Reference Manual (for Lisp Machines).
Computation Structures Group Memo 280, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1987.

Abstract: Id World is a powerful, integrated programming environment produced by the Computation Structures Group at MIT's Laboratory for Computer Science for the experimenter who wishes to prepare and execute parallel programs and study their behavior, as quickly as possible. Id World provides a powerful and convenient means to prototype parallel programs rapidly and to study these questions. Programs are written in the parallel programming language Id Nouveau and executed on GITA, an emulation of the MIT Tagged-Token Dataflow Architecture. GITA is heavily instrumented to measure many interesting aspects of parallelism.

Keywords: Dataflow, Id, Performance Monitoring, Prototyping.

[Nikhil, 1988]
Rishiyur S. Nikhil.
ID (Version 88.0) Reference Manual.
Computation Structures Group Memo 284, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1988.

Abstract: Id is a parallel programming language designed by members of the Computation Structures Group of MIT/LCS. It is used for programming dataflow and other parallel machines. Id is a functional language augmented with a parallel data-structuring mechanism called *I-structures*.

Keywords: Dataflow, Id.

[Nikhil, 1989]
R. Nikhil.
Can Dataflow Subsume von Neumann Computing?
In *15th Annual International Symposium on Computer Architecture*, page 131, Honolulu, Hawai, 1989
Keywords: Dataflow, Multi-threaded Architectures.

[Nöcker *et al.*, 1991a]
E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. Eekelen, and M. J. Plasmeijer.
Concurrent Clean.
In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE '91, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms*, volume 505 of *Lecture Notes in Computer Science*, pages 202–219, Eindhoven, The Netherlands, June 10–13, 1991. Springer, Berlin.

Abstract: Concurrent Clean is an experimental, lazy, higher-order parallel functional programming language based on term graph rewriting. An important difference with other languages is that in Clean graphs are manipulated and not terms. This can be used by the programmer to control communication and sharing of computation. Cyclic structures can be defined. Concurrent Clean furthermore allows to control the (parallel) order of evaluation to make efficient evaluation possible. With help of sequential annotations the default lazy evaluation can locally be changed into eager evaluation. The language enables the definition of partially strict data structures which make a whole new class of algorithms feasible in a functional language. A powerful and fast strictness analyzer is incorporated in the system. The quality of the code generated by the Clean compiler has been greatly improved such that it is one of the best code generators for a lazy functional language. Two very powerful parallel annotations enable the programmer to define concurrent functional programs with arbitrary process topologies. Concurrent Clean is set up in such a way that the efficiency achieved for the sequential case can largely be maintained for a parallel implementation on loosely coupled parallel machine architectures.

Keywords: Para-Functional Programming, Annotations, Strictness Analysis.

[Nöcker *et al.*, 1991b]
Eric Nöcker, Rinus Plasmeijer, and Sjaak Smetsers.
The Parallel ABC Machine.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 351–381, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: In this paper we will introduce the parallel ABC machine. The parallel ABC machine is an abstract parallel machine used for creating and describing implementations of Concurrent Clean, a parallel functional language based on the principles of term graph rewriting. Basically, the parallel ABC machine represents a loosely coupled processor architecture, but abstracts from specific implementation issues. We will show how such a machine can be defined, and how code for it can be generated. As for the sequential ABC machine, the description will be given in a functional framework.

Keywords: Term Graph Rewriting, Abstract Machine.

[Nöcker, 1993]
Eric Nöcker.
Strictness Analysis using Abstract Reduction.
In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 210–219, Copenhagen, Denmark, June 9–11, 1993. ACM Press.

Abstract: In this paper we present a new and general strictness analysis technique for lazy functional languages. In contrast to many other methods, this method is practically usable. This is shown with results of a real implementation. The key idea is the use of an abstract domain of which the elements represent various kinds of sets of concrete values. Reduction as well as pattern matching in the abstract domain mimics reduction reduction and pattern matching in the concrete domain in a very natural way. With this abstract domain various kinds of strictness analysis are possible. In particular, higher order functions and general data types (tuples and lists, but also user defined data types) fit perfectly well in this mechanism. In contrast to methods based on abstract interpretation, the abstract domain is infinite. Recursive functions are handled by a technique called reduction path analysis. Complicated and expensive fixed point derivation techniques are not needed. The implementation in the Concurrent Clean system shows that the analyser is very fust and that it finds much strictness information.

Keywords: Strictness Analysis, Concurrent Clean.

[Oldehoeft and Cann, 1988]
R. R. Oldehoeft and D. C. Cann.
Applicative Parallelism on a Shared-Memory Multiprocessor.
IEEE Software, 5(1):62–70, January 1988
Keywords: Parallel Functional Programming.

[Oldehoeft and McGraw, 1990]
R. R. Oldehoeft and J. R. McGraw.
Mixed Applicative and Imperative Programs.
Parallel Computing, 13(2):175–191, 1990.

Abstract: A modular organization for SISAL allows interface with conventional libraries.

Keywords: SISAL.

[Oldehoeft *et al.*, 1986]
R. R. Oldehoeft, D. C. Cann, and S. J. Allan.
SISAL: Initial MIMD Performance Results.
In *Conference on Algorithms and Hardware for Parallel Processing*, pages 120–127, Aachen, Germany, September 1986. Springer, Berlin
Keywords: Dataflow, Sisal, Compilation.

[Osborne, 1988]
Randy B. Osborne.
Speculative Computation in Multilisp.
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 103–137, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

Abstract: We demonstrate by experiments that performing computations in parallel before their results are known to be required can yield performance improvements over conventional approaches to parallel computing. We call such eager computation of expressions *speculative* computation, as opposed to conventional *mandatory* computation that is used in almost all contemporary parallel programming languages and systems. The two major requirements for speculative computation are: 1) a means to control computation to favor the most promising computation and 2) a means to abort computation and reclaim computation resources. We discuss these requirements in the parallel symbolic language Multilisp and present a *sponsor model* for speculative computation in Multilisp which handles control and reclamation of computation in a single, elegant framework. We describe an implementation of this sponsor model and present performance results for several applications of speculative computation. The results demonstrate that our support for speculative computation adds expressive and computational power to Multilisp, with observed performance improvement as great as 26 times over conventional approaches to parallel computation.

Keywords: Lisp, Speculative Parallelism.

[Ostheimer, 1991]
Gerrrald Ostheimer.
Parallel Functional Computation on STAR:DUST.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 393–407, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: STAR:DUST ('St. Andrews RISC: Dataflow Using Sequential Threads') is a processor design optimized for efficient execution of sequential threads while supporting plug-and-play construction of large multiprocessor systems. Besides satisfying the major RISC criteria (small instruction set, simple instruction format, load/store principle pipelining), STAR:DUST employs a dataflow approach to communication and parallelism. We will describe the architecture and propose a runtime model for parallel functional computation based on STAR:DUST's dataflow primitives.

Keywords: Dataflow, Sequential Threads, Hybrid Architecture.

[Padget and Merrall, 1991]
Julian Padget and Simon Merrall.
Bridging the MIMD–SIMD Gap.
In *Workshop on Abstract Machine Models for Parallel Computation*, 1991.

Abstract: This paper concerns some programming models for SIMD architectures to support symbolic processing. We report some initial experience with the MasPar array processor and the development of an implementation of EuLisp, extended by the paralation data structure as the first stages in this exercise. In particular, it seems that the paralation structure, although originally conceived for array architectures (viz. the Connection Machine), could be applied equally well to MIMD process management. We describe the notion of the active paralation as a mechanism to bridge the gap between SIMD and MIMD architectures.

Keywords: Lisp, SIMD, MIMD.

[Papadopoulos and Culler, 1990]
Gregory M. Papadopoulos and David E. Culler.
Monsoon: An Explicit Token-Store Architecture.
In *17th International Symposium on Computer Architecture*, number 18(2) in ACM SIGARCH Computer Architecture News, pages 82–91, Seattle, Washington, May 28–31, June 1990.

Abstract: Dataflow architectures tolerate long unpredictable communication delays and support generation and coordination of parallel activities directly in hardware, rather than assuming that program mappings will cause these issues to disappear. However, the proposed mechanisms are complex and introduce new mapping complications. This paper presents a greatly simplified approach to dataflow execution, called the *explicit token store* (ETS) architecture, and its current realization in *Monsoon*. The essence of dynamic dataflow execution is captured by a simple transition on state bits associated with storage local to a processor. Low-level storage management is performed by the compiler in assigning nodes to slots in an *activation frame*, rather than dynamically in hardware. The processor is simple, highly pipelined, and quite general. It may be viewed as a generalization of a fairly primitive von Neumann architecture. Although the addresssing capability is restrictive, there is exactly one instruction executed for each action on the dataflow graph. Thus, the machine oriented ETS model provides new understanding of the merits and the real cost of direct execution of dataflow graphs.

Keywords: Dataflow, Compilation, Abstract Machine.

[Papadopoulos, 1988]
Gregory Michael Papadopoulos.
Implementation of a General Purpose Dataflow Multiprocessor.
PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1988.

Abstract: General purpose multiprocessors have largely failed to meet expectations for programmability and performance. We blame the lack of usable parallel programming languages and systems on the underlying processor architecture. Machines built out of conventional sequential processors simply do not support the synchronization demands of parallel execution, so the programmer focuses upon the dangerous and arduous task of discovering a minimum set of synchronization points without introducing nondeterminism. We argue that processors must be fundamentally changed to execute a parallel machine language, in which parallel activities are coordinated as efficiently as instructions are scheduled. Dataflow architectures address this challenge by radically reformulating the basic specification of a machine program. These machines *directly execute* dataflow graphs, which specify only the essential prerequisites for the execution of an instruction — the availability of operands. Unfortunately, dataflow machines, including the M.I.T. Tagged Token Dataflow Architecture (TTDA), have labored under a number of implementation burdens, notably the apparent need for a fully associative operand matching store which discovers when instructions are able to execute. We introduce and develop a novel dataflow architecture, the Explicit Token Store (ETS), which directly executes tagged-token dataflow graphs while correcting a number of inherent inefficiencies of previous dataflow machines. In the ETS model, operand matching is performed at compiler-designated offsets within an activation frame. We show that the ETS is compatible with the TTDA by giving translations from TTDA machine graphs to ETS machine graphs. Finally, we describe an implementation of an ETS dataflow multiprocessor, called Monsoon, now under construction.

Keywords: Dataflow, Compilation, Id.

[Paulsen, 10983]
Christian Paulsen.
A Parallel Implementation of a Functional Language.
Technical report, Computer Science Department, Institute of Mathematics, University of Aarhus, Denmark, January 10983
Keywords: Parallel Functional Programming.

[Pehoushek and Weening, 1988]
Joseph D. Pehoushek and Joseph S. Weening.
Low-Cost Process Creation and Dynamic Partitioning in Qlisp.
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 182–199, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

Abstract: Our experiments with Qlisp have lead to two ways of improving the performance of parallel Lisp programs. One is to reduce the cost of process creation and scheduling; the other is to use a dynamic partitioning and scheduling method. We describe these techniques and present the results of several experiments that use them. We also present an analysis of the dynamic partitioning method to explain the reasons for its success.

Keywords: Lisp, Process Creation, Scheduling.

[Perry, 1988a]
Nigel Perry.
I/O and Inter-language Calling for Functional Languages.
Technical report, Department of Computing, Imperial College, London, UK, 1988.

Abstract: This paper discusses the provision of I/O and inter-language calling facilities in pure functional languages. In the introduction we first describe the scope of "I/O" by specifying a number of broad operation categories. We then describe why the "purity" of functional languages poses problems in providing "safe" I/O — compared to the same task in an imperative language. The first section is a survey existing I/O proposals for functional languages which fall into two groups; environment passing where the file system, I/O devices etc., are passed around as either an explicit or implicit abstract data structure by the program. Secondly stream based systems, where all I/O is channelled over "streams" which either carry basic values or more complex data types representing operation request to the O/S. In the second section we define a new system, termed *result continuations*, which provides a complete and safe I/O system. We show how all the operation categories are handled by this system and prove that the system is safe. The final section looks at providing inter-languages calling between functional and imperative languages. We show how, by using result continuations, an interface may be provided which does not compromise the referential transparency of the functional language.

Keywords: Operating Systems, Non-determinism, Result Continuations.

[Perry, 1988b]
Nigel Perry.
Towards a Functional Operating System.
Technical report, Department of Computing, Imperial College, London, UK, 1988.

Abstract: In this short note we discuss an operating system design based around our previously proposed *result continuations* thus avoiding the problems of stream I/O. To handle non-determinism a mall preemptive process scheduler is defined which is written in assembler and *not* a functional language, thus avoiding introducing a non-referentially transparent construct into the language. This two tiered approach is not unusual for an operating system written in a high-level language, c.f. Unix.

Keywords: Operating Systems, Non-Determinism, Result Continuations.

[Perry, 1992]
Nigel Perry.
Towards a Concurrent Object/Process Oriented Functional Language (Short Version).
In *15th Australian Computer Science Conference*, volume 14(1) of *Australian Computer Science Communications*, January 1992.

Abstract: In this paper we first define the programming methodology continuation passing style (CSP), which originates from tail functions and upon which result continuations are based, and then give an introduction to functional language I/O and inter-language working. We then show how the result continuation mechanism can be extended to support coroutine programming, and then use this to define remote function call (RPC) and object style systems. Next we show how concurrent functional programming can be supported using the result continuation mechanism, this work stems from our sketch of a functional operating system described in Perry. In conclusion we describe how this paper lays the groundwork for the development of concurrent functional and object/process-oriented programming paradigms and support non-determinism. A semantics for result continuations, functional coroutines, remote function calls, and concurrent functional programming is being developed. The semantics are defined using a version of the π-calculus and enable concurrent functional programs to be reasoned about. This reasoning can either be at two levels; using standard functional notation (λ-calculus) reasoning for the individual processes and π-calculus style reasoning for the concurrent communication, treating each process as a "black box"; or at a single level by implementing the λ-calculus in the π-calculus.

Keywords: Continuation Passing, Non-Determinism, Concurrent Objects, Parallel Semantics.

[Pettorossi, 1981]
A. Pettorossi.
An Approach to Communications and Parallelism in Applicative Languages.
In I. Ramos, editor, *Formalization of Programming Concepts*, pages 432–446. Springer, Berlin, 1981
Keywords: Parallel Functional Programming.

[Peyton Jones and Lester, 1992]
Simon L. Peyton Jones and David R. Lester.
Implementing Functional Languages: a tutorial.
Prentice Hall, 1992.

Abstract: This book gives a practical approach to understanding implementations of non-strict functional languages using lazy graph reduction. The book is intended to be a source of practical labwork material, to help make functional-language implementations "come alive", by helping the reader to develop, modify and experiment with some non-trivial compilers. The unusual aspect of the book is that it is meant to be *executed* as well as *read*. Rather than merely presenting an abstract description of each implementation technique, we present the code for a complete working prototype of each major method, and then work through a series of improvements to it. All of the code is available in machine-readable form. A section of the book is devoted to the development of an abstract machine and a compiler for a parallel G-machine.

Keywords: Parallel Graph Reduction.

[Peyton Jones and Salkild, 1989]
Simon L. Peyton Jones and Jon Salkild.
The Spineless Tagless G-machine.
In *FPCA '89, Functional Programming Languages and Computer Architecture*, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: The Spineless Tagless G-machine is an abstract machine based on graph reduction, designed as a target for compilers for non-strict functional languages. As its name implies, it is a development of earlier work, especially the G-Machine and TIM. It has a number of unusual features: the abstract machine code is rather high-level than is common, allowing better code generation; the representation of the graph eliminates most interpretive overheads; vectored returns from data structures give fast case-analysis; and the machine is readily extended for a parallel implementation.

Keywords: Parallel Graph Reduction, G-Machine.

[Peyton Jones *et al.*, 1989]
S. L. Peyton Jones, C. Clack, and J. Salkid.
High-Performance Parallel Graph Reduction.
In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, volume 365 of *Lecture Notes in Computer Science*, pages 193–206, Eindhoven, The Netherlands, June 12–16, 1989. Springer, Berlin.

Abstract: Parallel graph reduction is an attractive implementation for functional programming languages becaue of its simplicity and inherently distributed nature. This paper outlines some of the issues raised by parallel compiled graph reduction, and presents the approach we have adopted for our parallel machine, GRIP. We concentrate on two main areas: Static and dynamic techniques to control the growth of parallelism, so as to provide enough parallelism of an appropriate granularity to keep the machine busy without swamping it. Dynamic techniques to exploit the memory hierarchy, so that frequently-referenced data is held near to the processor that references it.

Keywords: Parallel Graph Reduction, Growth of Parallelism.

[Peyton Jones, 1985]
Simon L. Peyton Jones.
Generating Parallelism from Strictness Analysis.
Internal Note 1679, Department of Computer Science, University College, London, UK, 1985.

Abstract: The author investigates the use of strictness analysis to determine at compile time which parts of program evaluation can be carried out simultaneously. He gives a practical explanation of this technique and presents measurements of the actual parallelism produced.

Keywords: Strictness Analysis, Parallelization.

[Peyton Jones, 1987]
Simon L. Peyton Jones.
The Implementation of Functional Programming Languages.
International Series in Computer Science, Prentice-Hall, New York, 1987.

Abstract: This standard work on the implementation of functional languages by graph reduction also contains a section on parallel graph reduction discussing the issues of conservative/speculative parallelism, task granularity and task locality.

Keywords: Parallel Graph Reduction.

[Peyton Jones, 1989]
Simon L. Peyton Jones.
Parallel Implementations of Functional Programming Languages.
The Computer Journal, 32(2):175–186, April 1989.

Abstract: One of the most attractive features of functional programming languages is their suitability for programming parallel computers. This paper is devoted to discussion of such a claim. Firstly, parallel functional programming is discussed from the programmer's point of view. Secondly, since most parallel functional language implementations are based on the concept of graph reduction, the issues raised by graph reduction are discussed. Finally, the paper concludes with a case study of a particular parallel graph reduction machine and a survey of other parallel architectures.

Keywords: Parallel Graph Reduction, Para-Functional Programming, Dataflow.

[Peyton Jones, 1991]
Simon L. Peyton Jones.
The Spineless Tagless G-machine: Second Attempt.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 147–192, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: The Spineless Tagless G-Machine is an abstract machine designed to support functional languages. This presentation of the machine falls into two parts. Firstly, we present the *STG language*, an austere but recognizably-functional language, which as well as a *denotational* meaning has a well-defined *operational* semantics. The STG language is the "abstract machine code" for the Spineless Tagless G-machine, but it is sufficiently abstract that it can readily be compiled into G-machine G-code or TIM code instead. Secondly, we discuss the mapping of the STG language onto stock hardware. The success of an abstract machine model depends largely on how efficient this mapping can be made, though this topic is often relegated to a short section. Instead, we give a detailed discussion of the design issues and the choices we have made. Our principal target is the C language, treating the C compiler as a portable assembler.

Keywords: Graph Reduction, Abstract Machine.

[Peyton Jones, 1992]
Simon L. Peyton Jones.
Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine.
Journal of Functional Programming, 2(2):127–202, April 1992.

Abstract: The Spineless Tagless G-Machine is an abstract machine designed to support higher-order functional languages. This presentation of the machine falls into three parts. Firstly, we give a general discussion of the design issues involved in implementing non-strict functional languages. Next, we present the *STG language*, an austere but recognizably-functional language, which as well as a *denotational* meaning has a well-defined *operational* semantics. The STG language is the "abstract machine code" for the Spineless Tagless G-machine. Lastly, we discuss the mapping of the STG language onto stock hardware. The success of an abstract machine model depends largely on how efficient this mapping can be made, though this topic is often relegated to a short section. Instead, we give a detailed discussion of the design issues and the choices we have made. Our principal target is the C language, treating the C compiler as a portable assembler.

Keywords: Parallel Graph Reduction, Abstract Machine.

[Pingali, 1990]
Keshav K. Pingali.
Lazy Evaluation and the Logic Variable.
In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 7, pages 171–198, Addison-Wesley, Reading, Massachusetts, 1990.

Abstract: Functional languages can be enriched with logic variables to provide new computational features such as incremental construction of data structures. In this paper, we present a novel application for logic variables that highlights their importance: we argue that they are essential for efficient implementations of functional languages. This point is made by demonstrating that logic variables are required for explicating the process of demand propagation in lazy evaluation of functional programs. There are two applications of this result. For dataflow researchers, it offers a simple and efficient implementation of laziness on dataflow machines. For researchers investigating lazy graph reduction, it suggests new strictness analysis algorithms in which logic variables play an important role.

Keywords: Logic Variables, Strictness Analysis.

[Prini, 1980]
G. Prini.
Explicit Parallelism in LISP-like Languages.
In *1980 LISP Conference*, pages 13–18, Stanford, California, August 25–27, 1980.

Abstract: We introduce a LISP-like language whose parameter passing mechanism and control primitives allow for the creation and the synchronization of an arbitrary number of concurrent computations. The parameter passing mechanism is a parallel version of call-by-need: an argument of a function is evaluated only the first time the value of the corresponding formal parameter is needed during the evaluation of the function body, but such an evaluation is only *initiated* by-need and is then performed in parallel with all the other computations. Some primitives allow the user to initiate new computations *explicitly* and to check/wait for the termination of already initiated computations. Several examples illustrate how classical problems in multiprogramming have natural and elegant solutions using parallel call-by-need and the above mentioned primitives.

Keywords: Lisp, Parallel Lazy Evaluation, Streams.

[Rabhi and Manson, 1991]
F. A. Rabhi and G. A. Manson.
Divide-and-Conquer and Parallel Graph Reduction.
Parallel Computing, 17:189–205, 1991.

Abstract: This reference describes a mechanism of controlling the execution of functional programs with a Divide-and-Conquer type of parallelism.

Keywords: Parallel Graph Reduction, Program Control.

[Rabhi and Schwarz, 1993]
F. W. Rabhi and J. Schwarz.
"Paradigm-Oriented" Design of Parallel Iterative Programs Using Functional Languages.
In C. A. Brebbia and H. Power, editors, *Applications of Supercomputers in Engineering III*, Computational Mechanics Publications, Elsevier Applied Science, 1993.

Abstract: Functional languages offer a high degree of abstraction to the progammer while containing a great deal of implicit parallelism. This parallelism could be efficiently exploited if *parallel algorithmic structures* were used in the design of algorithms. A structure captures the behaviour of a parallel programming paradigm and acts as a template in the design of an algorithm. This paper addresses the issue of defining a structure for static iterative transformation (SIT) algorithms which are coarse-grained data parallel algorithms with an iterative control structure. The parameters required by the structure are supplied by the programmer using a functional language, forming the problem specification. This specification can then be successively turned into a sequential functional program, then into a parallel program for a graph reduction machine, and finally into a program that maps on a specific parallel architecture.

Keywords: Algorithmic Skeletons, Parallel Graph Reduction.

[Rabhi, 1991]
Fethi A. Rabhi.
Experiments with a Transputer-Based Parallel Graph Reduction Machine.
Concurrency: Practice and Experience, 3(4):413–422, August 1991.

Abstract: This paper is concerned with the implementation of functional languages on a parallel architecture, using graph reduction as a model of computation. Parallelism in such systems is automatically derived by the compiler but a major problem is the fine granularity, illustrated in Divide-and-Conquer problems at the leaves of the computational tree. The paper addresses this issue and proposes a method based on static analysis combined with run-time tests to remove the excess in parallelism. We report experiments on a prototype machine, simulated on several connected INMOS transputers. Performance figures show the benefits in adopting the method and the difficulty of automatically deriving the optimum partitioning due to differences among the problems.

Keywords: Parallel Graph Reduction, Granularity Control.

[Rabhi, 1992]
F. A. Rabhi.

Run-time Control of the Granularity in Functional Languages.
In W. Joosen and E. Milgrom, editors, *European Workshop on Parallel Computing*, pages 356–359, Barcelona, Spain, March, 1992. IOS Press.

Abstract: This reference examines the issue of controlling the granularity of programs according to the sequential complexity of tasks.

Keywords: Granularity Control.

[Rabhi, 1993a]
F. A. Rabhi.
Exploiting Parallelism in Functional Languages: A "Paradigm-Oriented" Approach.
In T. Lake and P. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, Oxford University Press, 1993.

Abstract: Deriving parallelism automatically from functional programs is simple in theory but very few practical implementations have been realised. Programs may contain too little or too much parallelism causing a degradation in perrformance. Such parallelism could be more efficiently controlled if *parallel algorithmic structures* (or skeletons) are used in the design of algorithms. A structure captures the behaviour of a *parallel programming paradigm* and acts as a template in the design of an algorithm. This paper presents some important parallel programming paradigms and defines a structure for each of these paradigms. The iterative transformation paradigm (or geometric parallelism) is discussed in detail and a framework under which programs can be developed and transformed into efficient and portable implementations is presented.

.

[Rabhi, 1993b]
Fethi A. Rabhi.
Functional Languages and Parallelism.
In *3rd Workshop on Parallel Computing*, Polytechnique de Mons, Belgium, September 23–24, 1993.

Abstract: This document explains the relation between functional languages and parallelism and presents some of the techniques used to execute functional programs on multi-processors without intervention from the programmer. It is intended to be an introduction to non-functional programmers and contains 100 references for readers interested in a particular aspect. The document is divided as follows: BACKGROUND (introduces functional languages and their theoretical foundations), PARALLEL GRAPH REDUCTION (describes some of the issues related to evaluating functional programs in parallel), THE "PARADIGM-ORIENTED" SOLUTION (describes the relationship between parallel programming paradigms and functional languages and how it could be used to improve the performance of implementations).

Keywords: Survey, Parallel Graph Reduction, Skeletons.

[Reppy, 1989]
J. M. Reppy.
First-Class Synchronization Operations in Standard ML.
Technical Report 89-1068, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, 1989
Keywords: ML, Continuations.

[Reppy, 1991a]
J. H. Reppy.
CML: A Higher-Order Concurrent Language.
In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26(6) of *SIGPLAN Notices*, pages 294–305, Toronto, Ontario, Canada, June 26–28, 1991. ACM Press, New York.

Abstract: Concurrent ML (CML) is a high-level, high-performance language for concurrent programming. It is an extension of Standard ML (SML) and is implemented on top of Standard ML of New Jersey (SML/NJ). CML is a practical language and is being used to build real systems. It demonstrates that we need not sacrifice high-level notation in order to have a good performance.

Keywords: ML, Threads.

[Reppy, 1991b]
John H. Reppy.
Concurrent Programming with Events – The Concurrent ML Manual.
Reference Manual Version 0.9.6, Department of Computer Science, Cornell University, Ithaca, NY, October 1991.

Abstract: Concurrent ML (CML) is a system for concurrent programming in Standard ML (SML). A CML program consists of a set of *threads* (or light-weight processes). A thread is the sequential evaluation of a ML expression. It does not have to be a terminating computation; in fact, infinitely looping threads are often useful. The evaluation of a thread may involve communication with other threads, which is done by sending a message on a channel. Message passing is synchronous and forms the basis of communication and synchronization in CML. This model is extended by *first-class synchronous operations*, which provide a mechanism for building new synchronization and communication abstractions.

Keywords: ML, Threads.

[Richards Jr., 1984]
Hamilton Richards Jr.
An Applicative Programming Languages Bibliography.
Technical report, Burroughs Corporation, Austin Research Center, Austin, Texas, October 1984
Keywords: Bibliography.

[Roe, 1989]
Paul Roe.
Some Ideas on Parallel Functional Programming.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 338–352, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

Abstract: This paper argues that parallel functional programs must be efficient in order to achieve speed-up on parallel machines. This to end parallelism expression and parallelism efficiency are discussed. Different classes of parallel algorithms and their expression are considered, along with two particular sources of inefficiency: task size and expression re-evaluation

Keywords: Parallel Functional Programming.

[Roh *et al.*, 1993]
Lucas Roh, Walid A. Najjar, and A. P. Wim Böhm.
Generation and Quantitative Evaluation of Dataflow Clusters.
In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 159–168, Copenhagen, Denmark, June 9–11, 1993. ACM Press.

Abstract: Mulithreaded or hybrid von Neumann/dataflow execution models have an advantage over the fine-grain dataflow model in that they significantly reduce the run time overhead incurred by matching. In this paperr, we look at two issues related to the evaluation of a coarse-grain dataflow model of execution. The first issue concerns the compilation into a coarse-grain code from a fine-grain one. In this study, the concept of coarse-grain code is captured by *clusters* which can be thought of as mini-dataflow graphs which execute strictly, deterministically and without blocking. We look at two bottom-up algorithms: the *basic block* and the *dependence sets* methods, to partition dataflow graphs into clusters. The second issue is the actual performance of the cluster-based execution as several architecture parameters are varied (e.g. number of processors, matching costs, network latency, etc.). From the extensive simulation data we evaluate (1) the potential speedup over the fine-grain execution and (2) the effects of the various architecture parameterrs on the coarse-grain execution time, allowing us to draw conclusions on their effectiveness. The results indicate that even with a simple bottom-up algorithm for generating clusters, cluster execution offers a good speedup over the fine-grain execution over a wide range of architectures. They also indicate that coarse-grain execution is scalable, tolerrates network latency and high matching cost well; it can benefit from a higher output bandwidth of a processor and finally, a simple superscalar processor with the issue rate of two is sufficient to exploit the internal parallelism of a cluster.

Keywords: Dataflow, Compilation, Partitioning.

[Rosendahl, 1986]
Mads Rosendahl.
Automatic Program Analysis.
Master's thesis, Institute of Datalogy, University of Copenhagen, Denmark, 1986
Keywords: Granularity Analysis.

[Rosendahl, 1989]
Mads Rosendahl.
Automatic Complexity Analysis.
In *FPCA '89 Functional Programming Languages and Computer Architecture*, pages 144–156, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: One way to analyze programs is to derive expressions for their computational behavior. A time bound function (or worst-case complexity) gives an upper bound for the computation time as a function of the size of the input. We describe a system to derive such time bounds automatically using abstract interpretation. The semantics-based setting makes it possible to prove the correctness of the time-bound function. The system can analyze programs in a first-order subset of Lisp and we show how the system also can be used to analyze programs in other languages.

Keywords: Complexity Analysis, Abstract Interpretation.

[Ruggerio and Sargeant, 1987]
Carlos A. Ruggerio and John Sargeant.
Control of Parallelism in the Manchester Dataflow Machine.
In Gilles Kahn, editor, *FPCA '87, Functional Programming Language and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 1–15, Portland, Oregon, September 14–16, 1987. Springer, Berlin.

Abstract: Fine-grain parallel machines, such as tagged-token dataflow machines, allow very high degrees of program parallelism to be exploited for many applications. In fact, so much parallelism can be generated that it is necessary to *control parallelism* in order to bound storage use. This paper reviews software mechanisms for parallelism control, which rely on merely planting extra code to control execution order. Such methods are found to be inadequate, so a fundamental architectural mechanism known as throttle is considered necessary. Various attempts to design a throttle for the Manchester Dataflow Machine are described. The eventual solution, a coarse-grain, process-based throttle, is explained, and simulation results are presented which demonstrate its effectiveness.

Keywords: Parallelism Control.

[Sabot, 1988]
Gary Sabot.
The Paralation Model: Architecture Independent SIMD Programming.
MIT Press, Cambridge, MA, 1988
Keywords: SIMD, Lisp, Data-Parallelism.

[Sands, 1989]
David Sands.
Complexity Analysis for a Lazy Higher-Order Language.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 56–79, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

Abstract: This paper is concerned with the time-analysis of functional programs. Techniques which enable us to reason formally about a program's execution costs have had relatively little attention in the study of functional programming. We concentrate here on the construction of equations which compute the time-complexity of expressions in a lazy higher-order language. The problem with higher-order functions is that complexity is dependent on the cost of applying functional parameters. Structures called *cost closures* are introduced to allow us to model both functional parameters *and* the cost of their application. The problem with laziness is that complexity is dependent on *context*. Projections are used to characterise the context in which an expression is evaluated, and cost-equations are parameterized by this context-description to give a compositional time-analysis. Using this form of context information we introduce two types of time-equation: *sufficient-time* equations and *necessary-time* equations, which together provide bounds on the exact time-complexity.

Keywords: Granularity Analysis.

[Sargeant and Watson, 1991]
John Sargeant and Ian Watson.
Some Experiments in Controlling the Dynamic Behavior of Parallel Functional Programs.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 103–122, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: In our quest for the "Holy Grail" of efficient implicit parallelism, we have opted for a conventional architecture with physically distributed but virtually shared memory (the EDS machine), and a Large Grain Graph Rewriting computational model. On programming languages, we hedge our bets, but the work described here uses a (strict) functional language. This set of choices eliminates many of the problems and brings those which remain into sharp focus. "All" we need to do is to achieve efficient dynamic load balancing, and high granularity, and good data structure locality. We have a compiler from Hope+ to the EDS machine (simulator), and this way used to investigate the properties of a "real" parallel program, a relational database implementation. This exercise showed the inadequacy of static analysis alone for detecting "good" tasks to do in parallel. Attempts to overcome this by programmer annotations had limited success. Furthermore, experiments with a much simpler program showed alarming variations in performance due to apparently innocuous differences in the basic low-level mechanisms for spawning parallel tasks. We are investigating an approach whereby the program is first compiled with monitoring code inserted, and run (using smallish data) to produce statistics which are fed back into the compiler which then produces optimized parallel code. This work is embryonic, but the method seems to have considerable advantages over static analysis or hand annotation, and also provides extra performance information for the programmer.

Keywords: Load Balancing, Static analysis, Annotations, Monitoring.

[Sarkar and Cann, 1990]
V. Sarkar and D. Cann.
POSC — A Partitioning and Optimizing SISAL Compiler.
In *Proceedings of the ACM International Conference on Supercomputing*, June 1990
Keywords: Compilation, SISAL.

[Sarkar and Hennessy, 1986a]
Vivek Sarkar and John Hennessy.
Compile-time Partitioning and Scheduling of Parallel Programs.
In *SIGLAN '86 Symposium on Compiler Construction*, volume 21(7) of *SIGPLAN NOTICES*, pages 17–26, Palo Alto, California, June 25–27, 1986. ACM Press, New York.

Abstract: Partitioning and scheduling techniques are necessary to implement parallel languages on multiprocessors. Multiprocessor performance is maximized when parallelism between tasks is optimally traded off with communication and synchronization overhead. We present compile-time partitioning and scheduling techniques to achieve this tradeoff.

Keywords: Scheduling, Partitioning.

[Sarkar and Hennessy, 1986b]
Vivek Sarkar and John Hennessy.
Partitioning Parallel Programs for Macro-Dataflow.
In *1986 ACM Symposium on Lisp and Functional Programming*, pages 202–211, Cambridge, Massachusetts, August 4–6, 1986. ACM Press, New York.

Abstract: Partitioning techniques are necessary to execute functional programs at a coarse granularity. Fine granularity execution is inefficient on general purpose multiprocessors. There is a trade-off between parallelism and the overhead of exploiting parallelism. We present a compile-time partitioning approach to achieve this trade-off.

Keywords: Dataflow, Partitioning.

[**Sarkar** *et al.*, **1988**]
V. Sarkar, S. Skedzielewski, and P. Miller.
An Automatically Partitioning Compiler for Sisal.
In *CONPAR 88*, volume B, pages 26–33, Manchester, UK, September 12–16, 1988. British Computer Society.

Abstract: In this paper, we describe a compiler that automatically repackages Sisal programs to achieve the grain size needed to run efficiently on different multiprocessors. The compiler is based on an existing implementation for Sisal on the Sequent Balance multiprocessor, and on previous work on automatic partitioning of Sisal programs.

Keywords: Partitioning, Compilation.

[**Sarkar, 1989**]
Vivek Sarkar.
Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.
Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989
Keywords: Parallelization, Compilation.

[**Schaefer and Schnoebelen, 1987**]
Peters Schaefer and Philippe Schnoebelen.
Specification of a Pipelined Event Driven Simulator using FP2.
In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87, Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures*, volume 258 of *Lecture Notes in Computer Science*, pages 311–328, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

Abstract: This paper deals with the specification of a pipelined event driven algorithm for discrete simulation using the Functional Parallel Programming Language FP2. We start with a short review of the pipelined event driven simulation algorithm and the main features of FP2. In the main part, the FP2 specification procedure and a proof methodology for liveness and correct termination of the specified network of processes are described.

Keywords: FP2, Process Networks, Verification.

[**Schauser** *et al.*, **1991**]
Klaus Erik Schauser, David E. Culler, and Thorsten von Eicken.
Compiler-Controlled Multithreading for Lenient Parallel Languages.
In *Functional Programming Languages and Computer Architectures*, volume 523 of *Lecture Notes in Computer Science*, pages 50–71, Harvard, Massachusetts, USA, 1991. Springer, Berlin.

Abstract: Tolerance to communication latency and inexpensive synchronization are critical for general-purpose computing on large multiprocessors. Fast dynamic scheduling is required for powerful non-strict parallel languages. However, machines that support rapid switching between multiple execution threads remain a design challenge. This paper explores how multithreaded execution can be addressed as a compilation problem, to achieve switching rates approaching what hardware mechanisms might provide. Compiler-controlled multithreading is examined through compilation of a lenient parallel language, Id90, for a threaded abstract machine, TAM. A key feature of TAM is that synchronization is explicit and occurs only at the start of a thread, so that a simple model can be applied. A scheduling hierarchy allows the compiler to schedule logically related threads closely together in time and to use registers across threads. Remote communication is via message sends and split-phase memory accesses. Messages and memory replies are received by compiler-generated message handlers which rapidly integrate these events with thread scheduling. To compile Id90 for TAM, we employ a new parallel intermediate form, dual-graphs, with distinct control and data arcs. This provides a clean framework for partitioning the program into threads, scheduling threads, and managing registers under asynchronous execution. The compilation process is described and preliminary measurements of its effectiveness are discussed. Dynamic execution measurements are obtained via a second compilation step, which translates TAM into native code for existing machines with instrumentation incorporated. These measurements show that the cost of compiler-controlled multithreading is within a small factor of the cost of control flow in sequential languages.

Keywords: Lenient Languages, Scheduling, Target Language.

[Schepers, 1991]
Jörg Schepers.
Using Functional Languages for Process Specifications.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 89–102, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: The GRAPH-system, a tool for the specification, analysis and execution of communicating functional processes, is presented. Processes are specified in GRAPH at two separate specification levels: At the functional level the input/output behavior of processes is specified by means of pgorgrams of a functional language. At the process level these processes constitute the atomic components of hierarchically structured process systems, wherein process communication is realized via streams. Different process types are available, supporting the design of dynamically expanding process structures as well as of non-deterministic processes, such as semaphores. Process systems are externally represented as graphs which are variants of Petri-nets, composed from processes and streams. The Petri-net semantics allows to verify statically certain invariance properties which guarantee a stable overall system behavior. Structural deadlocks can also be detected by a static analysis. The GRAPH-system provides a syntax-oriented editor for the graphical specification of process systems. Furthermore, it includes several control and analysis facilities and supports the stepwise execution of processes under interactive control.

Keywords: Process networks, streams, Petri nets, deadlocks.

[Schmittgen *et al.*, 1986]
C. Schmittgen, A. Gerdts, J. Haumann, W. E. Kluge, and M. Woitass.
A system-supported workload balancing scheme for cooperating reduction machines.
In *19th Annual Hawaii International Conference on System Sciences*, volume 1, pages 67–77, 1986
Keywords: Load Balancing.

[Schreiner, 1990a]
Wolfgang Schreiner.
ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language.
Master's thesis, RISC-Linz, Johannes Kepler University, Linz, Austria, October 1990, Also: Technical Report 90-42.0, RISC-Linz, Johannes Kepler University, Linz, Austria. Also: Technical Report 91-1, Austrian Center for Parallel Computation.

Abstract: This thesis describes the design and the implementation of the ADAM Abstract Dataflow Machine and the functional programming language EVE that will be used for programming this machine. ADAM is an abstract parallel computer with a hybrid distributed memory architecture that combines features from register machines and from dataflow computers. Its processor has a rather conventional set of two-address instructions augmented by primitives for the creation and synchronization of parallel tasks that allow the simulation of dataflow behavior. Special memory units on each ADAM module are responsible for the management of non-strict data structures. EVE is a simple purely functional language that shows most of the features known in functional programming: strong typing, higher order functions, non-strict functions and data types. A prototype of the ADAM concept has been implemented on a multi-transputer system and shows very promising results.

Keywords: Abstract Machine, Dataflow, Controlflow.

[Schreiner, 1990b]
Wolfgang Schreiner.
The ADAM Abstract Dataflow Machine.
In Ulrich Furbach, Christian Suttner, and Bertram Fronhöfer, editors, *Parallelization in Inference Systems*, volume 590 of *Lecture Notes in Artificial Intelligence*, pages 270–290, Dagstuhl Castle, Germany, December 17–18, 1990. Springer, Berlin.

Abstract: This paper describes the design and implementation of the ADAM Abstract Dataflow Machine. This abstract machine has a hybrid distributed-memory architecture combining features from conventional register machines and from dataflow computers. The ADAM processor has a conventional two-address instruction set augmented by primitives for the creation and synchronization of parallel tasks that allow the simulation of dataflow behavior. Special memory units on each ADAM module are responsible for the management of non-strict data structures. The system will be programmed in the non-strict functional (dataflow) programming language EVE that will be compiled to ADAM machine code. A prototype implementation of the concept on a multi-transputer system shows promising speed-ups.

Keywords: Abstract Machine, Dataflow, Controlflow.

[Schreiner, 1991]
Wolfgang Schreiner.
ADAM — An Abstract Dataflow Machine and Its Transputer Implementation.
In *Distributed Memory Computing, 2nd European Conference, EDMCC2*, volume 487 of *Lecture Notes in Computer Science*, pages 392–401, Munich, Germany, April 22–24, 1991. Springer, Berlin, Also: Technical Report 90-53, RISC-Linz, Johannes Kepler University, Linz, Austria. Also: Technical Report 91-3, Austrian Center for Parallel Computation.

Abstract: This paper describes the design of the ADAM Abstract Dataflow Machine. This abstract machine has a hybrid distributed-memory architecture combining features from conventional register machines and from dataflow computers. The ADAM processor has a conventional instruction set augmented by primitives for the creation and synchronization of parallel tasks. Memory units on each ADAM module are responsible for the management of non-strict data structures. A prototype implementation of the concept on a multi-transputer system shows promising speed-ups.

Keywords: Abstract Machine, Dataflow, Controlflow.

[Schreiner, 1992]
Wolfgang Schreiner.
On the Automatic Parallelizaton of List-Based Functional Programs.
In *Third International Workshop on Compilers for Parallel Computers*, pages 44–57, Vienna, Austria, July 6–9, 1992. Austrian Center for Parallel Computation Technical Report Series ACPC/TR 92-8.

Abstract: We investigate the behavior of functional programs operating on list structures with respect to automatic parallelization. Many list-processing functions follow typical patterns of recursion that contain inherent *horizontal parallelism*, *data parallelism*, *vertical parallelism* and *stream parallelism*. Starting from a simple classification of list-processing functions, more and more general versions of functional patterns are presented and it is shown how they cover many concrete programs. The operational behavior of these functional patterns in a parallel environment is discussed and criteria are presented that allow the automatic classification of functional programs.

Keywords: Streams, Data Parallelism.

[Schreiner, 1993]
Wolfgang Schreiner.
Parallel Functional Programming — An Annotated Bibliography.
Technical report, RISC-Linz, Johannes Kepler University, Linz, Austria, March 1993.

Abstract: This bibliography cites and comments more than 350 publications on the parallel functional programming research of the last 15 years. It focuses on the software aspect of this area i.e. on languages, compile-time analysis techniques (in particular for strictness and weight analysis), code generation, and runtime systems. Excluded from this bibliography are publications on special architectures and on garbage collection unless they contain aspects interesting for above areas. Most bibliographic items are listed inclusive their full abstracts.

Keywords: Bibliography.

[Seward, 1992]
Julian Seward.
Polymorphic Strictness Analysis using Frontiers (Draft Version).
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: This paper shows how to implement sensible polymorphic strictness analysis using the Frontiers algorithm. A central notion is to only ever analyze each function once, at its simplest polymorphic isntance. Subsequent non-base uses of functions are dealt with by generalizing their simplest instance analyzes. This generalisation is done using an algorithm developed by Baraki, based on embedding-closure pairs. Compared with an alternative approach of expanding the program out into a collection of monomorphic instances, this technique is hundreds of times faster for realistic programs. There are some approximations involved, but these do not seem to have a detrimental effect on the overall result. The overall effect of this technology is to considerably expand the range of programs for which the Frontiers algorithm gives useful results reasonably quickly.

Keywords: Strictness Analysis, Abstract Interpretation, Fixed Points.

[Sharp *et al.*, 1991]
David Sharp, Peter Harrison, and John Darlington.
A Synthesis of a Dynamic Message-Passing Algorithm for Quicksort.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 53–74, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: We present a methodology for systematically synthesizing algorithms for *message passing parallel machines* and illustrate the technique by the formal derivation of a message passing sorting algorithm, based on Quicksort, that is, we claim, the fastest parallel quicksort algorithm having a complexity of $O(\log n * \log n)$ for a machine with n processors sorting an n element list. The algorithm is thus highly suitable for massively parallel machines such as the Thinking Machines Connection Machine.

Keywords: Prrogram Transformation.

[Sijtsma, 1988]
B.A. Sijtsma.
Verification and Derivation of Infinite-List Programs.
PhD thesis, Groningen University, The Netherlands, October 1988.

Abstract: We will discuss various aspects of infinite-list programs. More specifically, we will discuss infinite-list programs written in a functional programming language with lazy evaluation. Our aim is mainly the verification of existing infinite-list programs. We present three related methods for deriving infinite-list programs from specifications. The methods are such that the verification and derivation go hand in hand; no separate verification is needed. These methods are in fact theorems and can thus be used to verify existing infinite-list programs. We also give a method which can be only used in special cases. In these special cases also the one of the general methods could have been used, but the special method generally leads to far more efficient programs. Finally, we derive a program for a certain class of infinite-list problems. This derivation illustrates how the scope of the methods can be extended.

Keywords: Infinite Lists, Circular Programs, Verification.

[Sijtsma, 1989]
Ben A. Sijtsma.
On the Productivity of Recursive List Definitions.
ACM Transactions on Programming Languages and Systems, 11(4):633–649, October 1989.

Abstract: Several related notions of the productivity are presented for functional languages with lazy evaluation. The notion of productivity captures the idea of computability, of progress of infinite-list programs. If an infinite-list program is *productive*, then every element of the list can be computed in finite "time". These notions are used to study recursive list definitions, that is, lists defined by l where $l = fl$. Sufficient conditions are given in terms of the function f that either guarantee the productivity of the list or its unproductivity. Furthermore, a calculus is developed that can be used in verifying that lists defined by l *where* $l = fl$ are productive. The power and the usefulness of our theory are demonstrated by several nontrivial examples. Several observations are given in conclusion.

Keywords: Circular Programs.

[Skedzielewski and Glauert, 1985]
Stephen Skedzielewski and John Glauert.
IF1 — An Intermediate Form for Applicative Languages, Version 1.0.
Technical Report M-170, Lawrence Livermore National Laboratory, University of California, July 1985.

Abstract: The intent of IF1 is to define a language that can be the target of several compilers for applicative languages. We hope to be able to exchange programs translated from different sources via IF1, and share the rather limited resource of programs written in applicative languages. The language that we define is not sufficiently general to be used by implementors of all applicative languages; it is strongly based on the features of SISAL and VAL. It is our hope that IF1 will be sufficient to form the basis for other, more comprehensive intermediate forms for a larger set of applicative languages.

Keywords: Intermediate Language, SISAL.

[Skedzielewski and Welcome, 1985]
Stephen Skedzielewski and Michael Welcome.
Data Flow Graph Optimization in IF1.
In Jean-Pierre Jouannaud, editor, *FPCA '85, Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 17–34, Nancy, France, September 16–19, 1985. Springer, Berlin, Also: Technical Report UCRL-92122, Lawrence Livermore National Laboratory, University of California.

Abstract: Optimization techniques are as important when compiling dataflow languages as when compiling conventional languages. This paper describes work that has been done on optimizers for SISAL programs that have been translated into IF1 data flow graphs. It shows that conventional optimization algorithms can be easily and efficiently implemented for data flow graphs, and that the payoff for even simple optimizations can be significant.

Keywords: Compilaton, Dataflow, SISAL.

[Skedzielewski, 1991]
Stephen K. Skedzielewski.
Sisal.
In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 4, pages 105–158, ACM Press, New York, 1991.

Abstract: The Sisal language is designed to be a notation for expressing solutions to large-scale scientific problems. In contrast to conventional *imperative* languages, Sisal belongs to a class of languages called *applicative languages*. This chapter describes Sisal Version 1.2 which is described in the reference manual dated March, 1985. This chapter does not attempt to describe *all* the features of Sisal. Rather, it will use a subset of the language as it attempts to motivate the use of Sisal in writing programs to exploit parallel computers.

Keywords: Dataflow, Sisal.

[Søndergaard and Sestoft, 1988]
H. Søndergaard and P. Sestoft.
Non-Determinism in Functional Languages.
Technical Report 88-18, Department of Computer Science, University of Melbourne, Australia, 1988
Keywords: Non-Determinism.

[Sören, 1983]
Holström Sören.
PFL: A Functional Language for Parallel Programming and Its Implementation.
In *Declarative Programming Worshop at University College*, London, UK, April 11–13, 1983. Also: Technical Report 83.03 R, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden
Keywords: Parallel Functional Programming.

[Spier and Szymanski, 1990]
Kevin L. Spier and B. K. Szymanski.
Interprocess Analysis and Optimization in the Equational Language Compiler.

In H. Burkhart, editor, *CONPAR 90 — VAPP IV, Joint International Conference on Vector and Parallel Processing*, volume 457 of *Lecture Notes in Computer Science*, pages 324–335, Zurich, Switzerland, September 10–13, 1990. Springer, Berlin.

Abstract: Our approach to interprocess data dependence analysis is to extend a dependence graph beyond the scope of each process. The extended graph is then analyzed using the attribute propagation technique developed in the EPL compiler for individual process consistency checking. The derived dependences, called external, are specific to each use of a process in a particular computation. As such, they are kept separate from the process dependence graph. A process needs to be recompiled only if the changes in other process' definitions change in external dependences. The described algorithm, although cast in terms of the EPL data dependence graph can be adapted to other languages. The method of analyzing the implications of interprocess interactions in parallel and distributed computations may be of value to other system.

Keywords: Data Dependence Analysis.

[Steele Jr. and Hillis, 1986]
Guy L. Steele Jr. and W. Daniel Hillis.
Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing.
In *1986 ACM Conference on Lisp and Functional Programming*, pages 279–297, Cambridge, Massachusetts, August 4–6, 1986. ACM Press, New York.

Abstract: Connection Machine Lisp is a dialect of Lisp extended to allow a fine-grained, data-oriented style of parallel execution. We introduce a new data structure, the xapping, that is like a sparse array whose elements can be processed in parallel. This kind of processing is suitable for implementation by such fine-grained parallel computers as the Connection Machine System and NON-VON. Additional program notation is introduced to indicate various parallel operations. The symbols α and $\circ$ are used, in a manner syntactically reminiscent of the backquote notation in Common Lisp, to indicate what parts of an expression are to be executed in parallel. The symbol β is used to indicate permutation and reduction of sets of data. Connection Machine Lisp in practice leans heavilyon APL and FP and their descendants. Many ideas and stylistic idioms can be carried over directly. Some idioms of Connection Machine Lisp are difficult to render in APL because Connection Machine Lisp xappings may be sparse while APL vectors are not sparse. We give many small examples of programming in Connection Machine Lisp. Two metacircular interpreters for a subset of Connection Machine Lisp are presented. One is concise but suffers from defining α in terms of itself in such a way as to obscure its essential properties. The other is longer but facilitates presentation of these properties.

Keywords: Lisp, SIMD, Xappings.

[Stoye, 1984]
W. Stoye.
A New Scheme for Writing Functional Operating Systems.
Technical Report 56, University of Cambridge Computer Laboratory, Cambridge, UK, 1984.

Abstract: A scheme is described for writing nondeterministic programs in a functional language. The scheme is based on message passing between a number of expressions being evaluated in parallel.

Keywords: Operating Systems, Streams, Non-Determinism.

[Stoye, 1986]
William Stoye.
Message-based Functional Operating Systems.
Science of Computer Programming, 6(3):291–311, 1986
Keywords: Operating Systems.

[Swanson *et al.*, 1988]
M. Swanson, R. Kessler, and G. Lindstrom.
An Implementation of Portable Standard Lisp on the BBN Butterfly.
In *ACM Symposium on Lisp and Functional Programming*, pages 132–142, Snowbird, Utah, July 1988. ACM Press, New York
Keywords: Lisp, Parallelization.

[Syre, 1976]
J. C. Syre.
Parallelism Control and Synchronization in a Single Assignment Language.
In *Fourth Annual Computer Science Conference*, page 30, Anaheim, CA, 1976
Keywords: Parallelism Control.

[Szymanki, 1993]
Boleaslaw K. Szymanki.
Parallel Functional Language — EPL and Its Compiler.
Technical report, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York, October 1993.

Abstract: This is a review paper that discusses the research and development centered around the parallel functional language EPL — Equational Programming Language — and its compiler. The emphasis is on opportunities and challenges arising from the ue of functional paradigm for parallel processing. Our approach is based on a two-level computation description: individual process description in EPL and the macro data flow description called configuration. The configuration represents an overall computation as a set of interacting processes. The EPL compiler performs an extensive program analysis and customizes the generated parallel code to specific architectures. The parallel computation decomposition and synchronization is based on the source program annotations provided by the user and on the user's defined configurations. The paper concentrates on motivation and general description of the features of the language, the detailed technical results were discussed elsewhere. There is also a comparison of EPL to other parallel functional languages, in particular SISAL and Crystal. The paper provides a brief description of future research agenda for EPL. It ends with a discussion of the general properties required of parallel functional languages and an assessment of EPL from that point of view.

Keywords: EPL.

[Szymanski, 1987]
B. Szymanski.
Parallel Programming with Recurrent Equations.
International Journal of Supercomputing Applications, 1(2):74–74, 1987
Keywords: EPL.

[Szymanski, 1991a]
Boleslaw K. Szymanski.
EPL — Parallel Programming with Recurrent Equations.
In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 3, pages 51–104, ACM Press, New York, 1991.

Abstract: This chapter presents recent work on creating and implementing a simple language for programming parallel scientific and engineering computations that is based on a few basic principles and yet satisfies the postulates discussed above. In addition, the chapter illustrates how a parallelizing compiler for this language works. The language, called *Equational Programming Language* (EPL), has been developed at Rensselaer Polytechnic Institute by the author and a group of his graduate students.

Keywords: Dataflow, EPL, Annotations.

[Szymanski, 1991b]
Boleslaw K. Szymanski, editor.
Parallel Functional Languages and Compilers.
Frontier Series, ACM Press, New York, 1991.

Abstract: A collection of articles on parallel implementations of functional languages.

Keywords: Parallel Functional Programming.

[Takeichi, 1987]
Masato Takeichi.
Partial Parameterization Eliminates Multiple Traversals of Data Structures.
Acta Informatica, 24:57–77, 1987.

Abstract: The use of higher order functions in functional programming opens up the possibility of defining functions by partial parameterization, and lazy evaluation brings out a new approach in programming methodology. This paper describes a new transformation technique based on partial parameterization and fully lazy evaluation for eliminating multiple traversals of data structures. It uses no particular mechanisms in functional programming, whereas it transforms a wider class of programs into efficient ones than that proposed so far.

Keywords: Partial Parameterization, Circular Programs.

[Takeuchi, 1988]
Ikuo Takeuchi.
Concurrent Programming in TAO — Practice and Experience.
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 271–291, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

Abstract: This paper describes various aspects of the concurrent programming in the Lisp language and related problems, based on our practice and experience with a concurrent Lisp dialect TAO. TAO realizes a multiple process and multiple user environment of practical performance and significance on a single ELIS processor by its powerful concurrent primitives. Topics described in this paper include: a Lisp-style process management, sharing Lisp programs among processes and users, name space problems around symbol packages, concurrent primitives, concurrent program debugging, and some performance figures. Further extension of TAO towards to a multiple ELIS processor system is also discussed briefly.

Keywords: Lisp.

[Talcott, 1986]
Carolyn Talcott.
Derived Properties and Derived Programs.
Technical report, Stanford University, California, May 1986.

Abstract: This paper gives a general theory for externalizing internal properties. A program can be extended to a program that returns the computation track of the evaluation. The computation time can then be expressed as the length of the computation track. The method can be used to analyze other internal properties such as maximum stack depth and number of function calls and the theory makes it possible to prove such extended programs correct.

Keywords: Granularity Analysis.

[Thompson, 1990]
Simon Thompson.
Interactive Functional Programs.
In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 10, pages 249–286, Addison-Wesley, Reading, Massachusetts, 1990.

Abstract: In this chapter, we present a model of interactive programs in a purely functional style. We exploit lazy evaluation in the modeling of streams as lazy lists. We show how programs may be constructed in an *ad hoc* way, and then we present as small set of interactions and combinators that form the basis for a disciplined approach to writing such programs. One of the difficulties of the *ad hoc* approach is that the way in which input and output are interleaved by the functions can be unpredictable. In the second half of the paper we use traces, i.e., partial histories of behavior, to explain the interleaving of input and output, and we give a formal explanation of our combinators. We argue that this justifies our claim that the combinators have the intuitively expected behavior, and finally, we contrast our approach with another.

Keywords: Process Networks, Streams, Non-determinism.

[Tick, 1988]
E. Tick.
Compile-Time Granularity Analysis for Parallel Logic Programming Languages.
In *International Conference on Fifth Generation Computer Systems*, pages 994–1000, Tokyo, Japan, November 28 – December 2, 1988. ICOT, Tokyo.

Abstract: The paper describes a simple compiler analysis method for determining the "weight" of procedures in parallel logic programming languages. Using Flat Guarded Horn Clauses (FGHC) as an example, the analysis algorithm is described. Consideration of weights has been incorporated in the scheduler of a real-parallel FGHC emulator running on the Sequent Symmetry multiprocessor. Alternative demand-distribution methods are discussed, including *oldest-first* and *heaviest-first* distributions. Performance measurements, collected from a group of non-trivial benchmarks on eight processors, show that the new schemes do *not* perform significantly faster than conventional distribution methods. This result is attributed to a combination of factors overshadowing the benefits of the new method: high system overheads, the low cost of spawning a goal on a shared memory multiprocessor, and the increase in synchronization caused by the new methods. Directions of further research are discussed, indicating where further speedup can be attained.

Keywords: Granularity Analysis.

[Traub, 1986]
Kenneth R. Traub.
A Compiler for the MIT Tagged-Token Dataflow Architecture.
Technical Report MIT/LCS/TR-370, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1986.

Abstract: Compilation of the programming language Id Nouveau into machine code for the MIT tagged-token dataflow architecture is thoroughly described. Id Nouveau is a higher-order functional language augmented with a novel data structure facility known as I-Structures. The tagged-token dataflow architecture is a dataflow computer of the dynamic variety. Compilation takes place in two steps. In the first step, the Id Nouveau program is converted into an abstract dataflow graph called a program graph. Program graphs embody no detailed knowledge of the target architecture, yet have a very precise operational semantics. At the same time, they represent data and control flow in a way very convenient for program transformation. Several common optimizing transformations are discussed. The second step of compilation converts the program graph into machine code for the tagged-token architecture, taking into account the machine's finite resources. Peephole optimizations for the machine code are discussed, and a general-purpose optimization algorithm is given.

Keywords: Compilation, Dataflow, Id.

[Traub, 1989]
Kenneth R. Traub.
Compilation as Partitioning: A New Approach to Compiling Non-Strict Functional Languages.
In *FPCA '89, Functional Programming Languages and Computer Architecture*, pages 75–88, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: In *non-strict* functional languages, a data structure may be read before all its components are written, and a function may return a value before finishing all its computation or even before all its arguments have been evaluated. Such flexibility gives expressive power to the programmer, but makes life difficult for the compiler because it may not be possible to totally order instructions at compile time; the correct order can very dramatically with the input data. Consequently, the compiler must break the program into sequential fragments, or *threads*, whose relative ordering is determined at run time. Good compilers employ strictness analysis and other techniques to make threads as large as possible, to minimize run-time overhead. While partitioning a program into sequential threads is a crucial issue, existing compilers treat it as a byproduct of applying some other methodology, such as performing Henderson's force/delay transformations or generating intermediate code for an abstract graph reduction machine (e.g., the G-machine or Tim). In this paper, we present a view of functional language compilation that takes partitioning a function into sequential threads as the first order of business. The resulting framework cleanly separates issues of partitioning, of thread scheduling, of environments, and of data type representation (including first-class functions). Our method sidesteps both the force/delay transformation and abstract machines, going directly from source code to sequential three-address code. Nevertheless, nearly all of the optimizations proposed for existing approaches are easily expressed. We consider two non-strict evaluation rules: the familiar lazy evaluation, and lenient evaluation as found in the language Id. We also consider both uniprocessor and parallel processor scheduling policies.

Keywords: Compilation, Thread Partitioning, Scheduling, Lenient Evaluation.

[Traub, 1991a]
Kenneth R. Traub.
Implementation of Non-Strict Functional Programming Languages.
Research Monographs in Parallel and Distributed Computing, Pitman, London, 1991.

Abstract: The authors' Ph.D. thesis introduces a new view on the compilation of functional languages with lenient (i.e. dataflow) semantics. The central task of the compiler is to partition a program into a set of sequential threads that execute sequentially and are scheduled by the runtime system. The details of the program analysis and the generation of the target code are described in large detail.

Keywords: Lenient Evaluation, Functional Quads, Dependence Analysis.

[Traub, 1991b]
Kenneth R. Traub.
Multi-thread Code Generation for Dataflow Architectures from Non-Strict Programs.
In John Hughes, editor, *Functional Programming Languages and Computer Architectures*, volume 523 of *Lecture Notes in Computer Science*, pages 73–101, Harvard, Massachusetts, USA, 1991. Springer, Berlin.

Abstract: This paper presents a new style of code generation for dataflow architectures, based on a view of such architectures as general multi-threaded von Neumann machines. Whereas the traditional picture of dataflow object code consists of tokens flowing along the arcs of a dataflow graph, the multi-threaded style treats a linear sequence of dataflow instructions as a sequential thread. Within a thread, data is passed by imperatively reading and writing an activation frame associated with a procedure invocation, just as in conventional architectures. Also, advanced dataflow architectures like Monsoon provide specific support for sequential threads, in the form of general registers for operand storage within a thread. Between threads, values are also communicated via the activation frame. The generation and synchronization of tokens, which in traditional dataflow is the primary means of communicating data, is relegated haere to a purely control flow role, spawning new threads and gating their initiation, but themselves carrying no data. Our results show that in many cases, the multi-threaded style of code generation results in fewer total machine cycles to executed a given program as compared to the traditional dataflow style. Surprisingly, this remains true even if the general registers are not employed. The reason the multi-threaded style is so successful is that by separating data flow from control flow, redundant control flow can be eliminated. In other words, fewer forking (token creation) and joining (token matching) operations are required. Central to eliminating redundant control flow is the compiler's ability to discover sequential threads that may be scheduled at compile time. While this is inherently difficult when starting from a non-strict language such as Id, even small threads have a beneficial effect. The most dramatic performance improvement comes about in compiling Id loop, where the language semantics specifies a greater degree of strictness, resulting in larger threads. We demonstrate the comparative performance between the multi-thread and traditional dataflow styles of code generation through empirical observations on the Monsoon hardware, and also through a detailed analytic examination of the code generated under each paradigm.

Keywords: Compilation, Multi-threading, Dataflow.

[Tsung-Min and Mishra, 1989]
Kuo Tsung-Min and Prateek Mishra.
Strictness Analysis: A New Perspective Based on Type Inference.
In *FPCA '89, Functional Programming Languages and Computer Architecture*, London, UK, September 11–13, 1989. ACM Press, New York.

Abstract: In this work we develop a new framework for strictness analysis. We provide a new definition of strictness based on operational semantics which directly addresses the primary aspect of the strictness analysis problem: non-termination of evaluation. Our definition excludes other irrelevant details such as the type system of the language, and imposes minimal constraints on semantic equality of expressions. We provide a new syntactic representation for strictness properties of programs and a new computational technique for strictness analysis. The syntactic representation takes the from of a simple type language for expressing strictness properties and is much simpler than representations used in previous work. The computational technique is based on well-understood type inference techniques and improves on previous work as it does not require fixpoint iteration. Our framework is parameterized with respect to the language of strictness properties and type inference techniques. As improvements in computational methods become available, both the language and type inference technique can be naturally extended to more complex and accurate strictness analysis.

Keywords: Strictness Analysis.

[Turner, 1981]
David Turner.
The Semantic Elegance of Applicative Languages.
In *FPCA '81, Functional Languages and Computer Architecture*, pages 85–92, Portsmouth, New Hampshire, October 18–22, 1981. ACM, New York.

Abstract: In what does the alleged superiority of applicative languages consist? In the last analysis the answer must be in terms of the reduction in the time required to produce a correct program to solve a given problem. On reflection I decided that the best way to demonstrate theis would be to take some reasonably non-trivial problem and show how, by proceeding within a certain kind of applicative language framework it was possible to develop a working solution with a fraction of the effort that would have been necessary in a conventional imperative language. The particular problem I have chosen also brings out a number of general points of interest which I shall briefly discuss afterwards.

Keywords: Non-strictness, Streams.

[Turner, 1987]
David Turner.
Functional Programming and Communicating Processes.
In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87 Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 54–74, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

Abstract: This paper reports some of the initial results of a research project at the University of Kent for the construction of an operating system written entirely in a functional language. We are here primarily concerned with the internal structure of the operating system, seen as a network of communicating processes. The description in purely functional terms of such a network has until recently posed logical problems, a solution to which was proposed by W. Stoye in 1984. We have developed a refinement of Stoye's model which has stronger synchronization properties and permits each process to run in a separate address space, facilitating a distributed implementation. By introducing the concept of a wrapper-function, we enable message passing code to be statically type checked.

Keywords: Operating System, Process Networks, Streams, Non-determinism.

[Turner, 1990]
David Turner.
An Approach to Functional Operating Systems.
In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 8, pages 199–218, Addison-Wesley, Reading, Massachusetts, 1990.

Abstract: This chapter reports some of the initial results of a research project at the University of Kent for the construction of an operating system written entirely in a functional language. We are here primarily concerned with the internal structure of the operating system, seen as a network of communicating processes. The description in purely functional terms of such a network has until recently posed logical problems, a solution to which was proposed by W. Stoye in 1984. We have developed a refinement of Stoye's model which has stronger synchronization properties and permits each process to run in a separate address space, facilitating a distributed implementation. By introducing the concept of a wrapper-function, we enable message passing code to be statically type checked.

Keywords: Operating System, Process Networks, Streams, Non-determinism.

[van Groningen, 1992]
John H. G. van Groningen.
Some Implementation Aspects of Concurrent Clean on Distributed Memory Architectures.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

Abstract: We have implemented a code generator and a runtime system that can be used to simulate parallel execution of Concurrent Clean programs on a single Macintosh computer. This code generator generates machine code and is an extension of the (sequential) ABC code generator for the MC680x0 and SPARC processors. This implementation is discussed briefly. The largest part of the paper describes two aspects of the implementation in detail. The first one is how to let a process wait until a node is overwritten with its head normal form. Several possible solutions are described. The other one is a description of an efficient graph copying algorithm. This graph copying algorithm can be used to send graphs to remote processors using a more compact representation. Finally, some results are presented.

Keywords: Parallel Graph Reduction.

[Vree, 1986]
Willem G. Vree.
The Grain Size of Parallel Computations in a Functional Program.
Technical report, Computer Science Department, Dutch Water Board Authority, Rijwijk, The Netherlands, 1986.

Abstract: One problem that limits the exploitability of the available parallelism in a functional language is that many function applications may not contain a sufficient amount of computation to justify the overhead that is caused by data transmission and process synchronization in parallel machine architectures. This paper concentrates on the grain size problem in relation to that class of MIMD architectures for which the data-communication and process-synchronization overrhead is large compared to the elementary machine instruction time. It will be demonstrated, with the aid of an example program of moderate size, that it is sometimes necessary to perform complicated program transformations in order to obtain a useful grain size for MIMD architectures.

Keywords: Grain Size.

[Wadge and Ashcroft, 1985]
William W. Wadge and Edward A. Ashcroft.
Lucid, the Dataflow Programming Language.
Number 22 in A.P.I.C. Studies in Data Processing, Academic Press, London, 1985.

Abstract: An extensive description of dataflow programming in Lucid.

Keywords: Dataflow, Lucid.

[Wadler and Hughes, 1987]
Philip Wadler and R. J. M. Hughes.
Projections for Strictness Analysis.
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

Abstract: Contexts have been proposed as a means of performing strictness analysis on non-flat domains. Roughly speaking, a *context* describes how much a sub-expression can be represented using the notion of *projection* from domain theory. This is clearer than the previous explanation of contexts in terms of continuations. In addition, this paper describes *finite domains* of contexts over the non-flat list domain. This means that recursive context equations can be solved using standard fixpoint techniques, instead of the algebraic manipulation previously used.

Keywords: Strictness Analysis, Projections.

[Wadler, 1986]
Philip Wadler.
A New Array Operation.

In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction, Proceedings of a Workshop*, volume 274 of *Lecture Notes in Computer Science*, pages 328–335, Santa Fe, New Mexico, September 29 – October 1, 1986. Springer, Berlin.

Abstract: Providing efficient operations on arrays in a functional language is problematic. No completely satisfactory solution has yet been found. This paper proposes a new solution, which is a variant on the "monolithic" approach to array operations. The new solution is also not completely satisfactory, but does does have advantages complementary to other proposals. It makes a class of programs easy to express, notably those involving the construction of histograms. It also allows for parallel implementations without the need to introduce non-determinism.

Keywords: Updatable Arrays, Non-determinsim.

[Wadler, 1987]
Philip Wadler.
Strictness Analysis in Non-Flat Domains by Abstract Interpretation.
In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, Ellis Horwood, 1987
Keywords: Abstract Interpretation.

[Wadler, 1988]
Philip Wadler.
Strictness Analysis Aids Time Analysis.
In *Principles of Programming Languages*, 1988.

Abstract: This paper describes a method of developing step-counting versions for programs under lazy evaluation. The transformation is rather trivial under eager evaluation but under lazy evaluation the computation time depends on the context. The step-counting version is constructed by extending the function with an extra argument to describe which part of the result is needed.

Keywords: Granularity Analysis.

[Wadler, 1990]
Philip Wadler.
Comprehending Monads.
In *ACM Conference on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM Press, New York.

Abstract: Category theorists invented *monads* in the 1960's to concisely express certain aspects of universal algebra. Functional programmers invented *list comprehensions* in the 1970's to concisely express certain programs involving lists. This paper shows how list comprehensions may be generalized to an arbitrary monad, and how the resulting programming feature can concisely express in a purely functional language some programs that manipulate state, handle exceptions, parse text, or invoke continuations. A new solution to the old problem of destructive array update is presented. No knowledge of category theory is assumed.

Keywords: Mondads, Aggregate Update.

[Wadler, 1992]
Philip Wadler.
The Essence of Functional Programming.
In *19th Annual Symposium on Principles of Programming Languages*, January 1992.

Abstract: This paper explores the use of monads to structure functional programs. No prior knowledge of monads or even category theory is required. Monads increase the ease with which programs may be modified. They can mimic the effect of impure features such as exceptions, state, and continuations; and also provide effects not easily achieved with such features. The types of a program reflect which effects occur. The first section is an extended example of the use of monads. A simple interpreter is modified to support various extra features: error messages, state, output, and non-deterministic choice. The second section describes the relation between monads and continuation-passing style. The third section sketches how monads are used in a compiler for Haskell that is written in Haskell.

Keywords: Monads, State.

[Walinsky and Banerjee, 1990]
Clifford Walinsky and Deb Banerjee.
A Functional Programming Language Compiler for Massively Parallel Computers.
In *ACM Symposium on Lisp and Functional Programming*, pages 131–138, Nice, France, June 27–29, 1990. ACM Press, New York.

Abstract: Functional programming languages remove programmers from low-level machine details, an important achievement when programming massively parallel systems. We present an overview of an FP compiler that generates programs capable of exploiting data-parallelism, a view of parallelism where distinct data elements reside on distinct processors and all processors execute a single instruction stream. To achieve this form of parallelism, FP's sequences are represented as arrays. This representation makes possible optimization techniques developed for APL compilers that compose routing functions at compile-time. These techniques are described succinctly by a set of axioms and inference rules. We demonstrate the optimizations by compiling several FP functions, obtaining optimal performance.

Keywords: FP, Data Parallelism, SIMD.

[Wand, 1980]
Mitchell Wand.
Continuation-Based Multiprocessing.
In *1980 LISP Conference*, pages 19–28, Stanford, California, August 25–27, 1980.

Abstract: Any multiprocessing facility must include three features: elementary exclusion, data protection, and process saving. While elementary exclusion must rest on some hardware facility (e.g. a test-and-set instruction), the other two requirements are fulfilled by features already presented in applicative languages. Data protection may be obtained through the use of procedures (closures or funargs), and process saving may be obtained through the use of the CATCH operator. The use of CATCH, in particular, allows an elegant treatment of process saving. We demonstrate these techniques by writing the kernel and some modules for a multiprocessing system. The kernel is very small. Many functions which one would normally expect to find inside the kernel are completely decentralized. We consider the implementation of other schedulers, interrupts and the implications of these ideas for language design.

Keywords: Lisp, Continuations, Operating Systems.

[Watson and Watson, 1986a]
Ian Watson and Paul Watson.
Graph Reduction in a Parallel Virtual Memory Environment.
In *Proceedings of the Graph Reduction Workshop*, Lecture Notes in Computer Science, pages 265–274, Santa Fe, New Mexico, September 29 – October 1, 1986. Springer, Berlin.

Abstract: This paper examines the area of memory organization and the mechanisms which are required to ensure that the full benefits of this implementation strategy are maintained whilst achieving an efficient physical structure. The major areas of interest are the maintenance of locality and the use of storage management techniques which minimize communication.

Keywords: Parallel Graph Reduction, Memory Management.

[Watson and Watson, 1986b]
Ian Watson and Paul Watson.
Parallel Data Driven Graph Reduction.
In J. V. Woods, editor, *Fifth Generation Computer Architectures*. Nort Holland, 1986
Keywords: Parallel Graph Reduction, Data Flow.

[Watson and Watson, 1987]
Paul Watson and Ian Watson.
Evaluating Functional Programs on the FLAGSHIP Machine.
In Gilles Kahn, editor, *FPCA '87, Functional Programming Language and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 80–97, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

Abstract: The Flagship project has the aim of designing a parallel computer system for the evaluation of declarative languages. The physical architecture of the machine consists of a set of closely coupled processor/store pairs communicating over a high performance communications network. A functional program to be evaluated is compiled into a Super-Combinator expression graph which is then simplified by graph reduction. The paper discusses the Flagship machine architecture and describes in detail the computational model which defines how a functional program is represented and reduced. The issues underlying the design of the main features of the computational models are examined.

Keywords: Parallel Graph Reduction.

[Watson, 1986]
Paul Watson.
The Parallel Reduction of Lambda Calculus Expressions.
PhD thesis, University of Manchester, UK, 1986.

Abstract: An approach to the parallel evaluation of functional programs represented as λ-expressions is described.

Keywords: Parallel Functional Programming, Lambda Calculus.

[Watson, 1989]
Ian Watson.
Simulation of a Physical EDS Machine Architecture.
Technical report, Department of Computer Science, University of Manchester, UK, September 1989.

[Waugh, 1991]
Kevin G. Waugh.
Parallel Imperative Programs from Functional Prototypes.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 75–88, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: We have been considering the automatic translation of sequential SML prototypes of vision algorithms into efficient parallel Occam programs. We use function instrumentation of the prototype to obtain typical and pathological execution details and static analysis allows us to consider argument sizes and flow. From these results an abstract form of the functional program is used to estimate partition size and hence granularity. The transformations which apply to the functional prototype can also to be applied to the abstract form of the program. In this way we can use the transformed abstracted program to determine the effect of particular transformations on execution of the prototype.

Keywords: Program Transformation, Weight Analysis.

[Weber, 1989]
Ken Weber.
dLisp — A LAN-Distributed Lisp System.
Technical report, Department of Mathematical Sciences, Kent State University, February 1989.

Abstract: dLisp, a package of functions that can be added to existing Lisp systems to provide for distributed evaluation of Lisp forms on a local area network, is described and defined. A few programming examples are given and a prototype implementation is discussed briefly.

Keywords: Lisp, Networks.

[Weening, 1988]
J. S. Weening.
A parallel lisp simulator.
Technical Report STAN-CS-88-1206, Department of Computer Science, Stanford University, CA, 1988.

Abstract: CSIM is a simulator for parallel Lisp, based on a continuation passing interpreter. It models a shared-memory multiprocessor executing programs written in Common Lisp, extended with several primitives for creating and controlling processes. This paper describes the structure of the simulator, measures its performance, and gives an example of its use with a parallel Lisp program.

Keywords: Lisp.

[Wegbreit, 1975]
Ben Wegbreit.
Mechanical Program Analysis.
Communications of the ACM, 18(9):528–539, September 1975.

Abstract: One means of analyzing program performance is by deriving closed-form expressions for their execution behavior. This paper discusses the mechanization of such analysis, and describes a system, Metric, which is able to analyze simple Lisp programs and produce, for example, closed-form expressions for their running time expressed in terms of size of input. This paper presents the reasons for mechanizing program analysis, describes the operation of Metric, explains its implementation, and discusses its limitations.

Keywords: Granularity Analysis.

[Welcome *et al.*, 1986]
M. L. Welcome, S. K. Skedzielewski, R. K. Yates, and J. E. Ranelletti.
IF2: An Applicative Language Intermediate Form with Explicit Memory Management.
Manual M-195, University of California Lawrence Livermore National Laboratory, November 1986.

Abstract: This defines an intermediate form that includes IF1 and has the capability for describing operations on storage.

Keywords: Intermediate Language.

[Wetherell, 1982]
C. Wetherell.
Error Data Values in the Data-Flow Language VAL.
ACM Transactions on Programming Languages and Systems, 4(2):226–238, April 1982
Keywords: Dataflow, Error Handling.

[Williams *et al.*, 1990]
M. C. Williams, J. L. Armstrong, S. R. Virding, and B. O. Dacker.
Implementing a Functional Language for Highly Parallel Real Time Applications.
In *8th SETSS, Florence, Italy, March 30 – April 1*, 1990
Keywords: Erlang.

[Wise, 1987]
David Wise.
Matrix Algebra and Applicative Programming.
In Gilles Kahn, editor, *FPCA '87, Functional Programming Language and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 134–153, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

Abstract: The broad problem of matrix algebra is taken up from the perspective of functional programming. A key question is how arrays should be represented in order to admit good implementations of well-known efficient algorithms, and whether functional architecture sheds any new light on these or other solutions. It relates directly to disarming the "aggregate update" problem. The major thesis is that 2^d-ary trees should be used to represent d-dimensional arrays; examples are matrix operations ($d = 2$), and a particularly interesting vector ($d = 1$) algorithm. Sparse and dense matrices are represented homogeneously, but at some overhead that appears tolerable; encouraging results are retrieved and extended. A Pivot Step algorithm is described which offers optimal stability at no extra cost for searching. The new results include proposed sparseness measures for matrices, improved performance of stable matrix inversion through repeated pivoting while deep within a matrix-tree (extendible to solving linear systems), and a clean matrix derivation of the vector algorithm for the fast Fourier transform. Running code is offered in the appendices. This work is particularly important because of the importance of this family of problems. Progress would be of simultaneous use in decomposing algorithms over traditional vector multiprocessors, as well as motivate practical interest in highly parallel functional architectures

Keywords: Distributed Data Structures. Aggregate Update.

[Wray and Fairbairn, 1989]
S. C. Wray and J. Fairbairn.

Non-strict Languages — Programming and Implementation.
The Computer Journal, 32(2):142–151, 1989.

Abstract: Non-strict evaluation improves the expressive power of functional languages at the expense of an apparent loss of efficiency. In this paper we give examples of this expressive power, taking as an example an interactive functional program and describing the programming techniques depending on non-strict evaluation which improved its design. Implementation methods for non-strict languages have delivered poor performance precisely when such programming techniques have been used. This need not be the case, however, and in the second part of the paper we describe Tim, a method of implementing non-strict languages for which the penalty for using lazy evaluation is very small.

Keywords: Non-Strictness, Streams, Circular Programs.

[Wray, 1986]
Stuart Charles Wray.
Implementation and Programming Techniques for Functional Languages.
PhD thesis, Computer Laboratory, University of Cambridge, England, June 1986, Technical Report No. 92.

Abstract: In this thesis I describe a new method of strictness analysis for lazily evaluated functional languages, and a method of code generation making use of the information provided by this analysis. I also describe techniques for practical programming in lazily evaluated functional languages, based on my experience of writing substantial functional programs. My new strictness analyzer is both faster and more powerful than that of Mycroft. It can be used on any program expressed as super-combinator definitions and it uses the additional classifications *absent* and *dangerous* as well as strict and lazy. This analyzer assumes that functional arguments to higher order functions are completely lazy. I describe an extension of my analyzer which discovers more strictness in the presence of higher-order functions, and I compare this with higher order analyzers based on Mycroft's work. I also describe an extension of my analyzer to lazy pairs and discuss strictness analyzers for lazy lists. Strictness analysis brings useful performance improvements for programs running on conventional machines. I have implemented my analyzer in a compiler for Ponder, a lazily evaluated functional language with polymorphic typing. Results are given, including the surprising result that higher order strictness analysis is no better than first order strictness analysis for speeding up real programs on conventional machines. I have written substantial programs in Ponder, and I describe in some detail the largest of these, which is about 2500 lines long. This program is an interactive spreadsheet using a mouse and bit-mapped display. I discuss programming techniques and practical problems facing functional languages with illustrative examples from programs I have written.

Keywords: Strictness Analysis, Abstract Interpretation.

[Young, 1989]
Jonathan Hood Young.
The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages.
Research report yaleu/dcs/rr-669, Department of Computer Science, Yale University, May 1989.

Abstract: This thesis describes the implementation of an optimizing compiler for the language ALFL. Like many other lazy languages, a significant overhead is incurred in order to preserve the lazy semantics of the language. The optimizations developed in this thesis are designed to eliminate this overhead in as many places as possible by converting lazy evaluation (call-by-need) to eager evaluation (call-by-value). Converting call-by-need to call-by-value is not always a safe transformation; it must be proven to be safe at each opportunity where it arises in the source program. The safety of the transformations in this thesis is proven correct using the technique of *abstract interpretation:* we show that certain non-standard semantic analyses stand in a particular relation to the standard semantics of our language. The semantic analysis developed in this thesis include *strictness analysis, termination analysis, operation count analysis,* and *collecting interpretations.* We then discuss several practical issues related to the integration of these semantic analyses into a compiler. Several new and old techniques to compute or approximate least fixpoints of semantic equations are presented. Finally, we document the semantics-based tools used to build the compiler, including packages for caching the results of functions, manipulating sets, and creating and updating environments, as well as a highly abstract and parameterized semantic evaluator which implements the fixpoint finding techniques mentioned above.

Keywords: Compilation, Abstract Interpretation, Strictness Analysis, Operation Count Analysis, Collecting Interpretations, Fixed Points.

[Zimmer, 1991a]
R. M. Zimmer.
Zur Pragmatik eines Operationalisierten λ-Kalkuels als Basis fuer Interaktive Reduktionssysteme (On the Pragmatics of an Operationalized λ-Calculus as a Basis of Interactive Reduction Systems) (In German).
PhD thesis, Gesellschaft für Mathematik und Datenverarbeitung, Oldenbourg, Germany, 1991, GMD Bericht 192
Keywords: Graph Reduction.

[Zimmer, 1991b]
Ralf Zimmer.
Reflections, the Curch-Rosser Property, and "Real" Applications in Functional Systems.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 331–349, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

Abstract: Modern functional systems are often claimed to be well suited for programming and using multiprocessor machines. Unfortunately, most of them do not feature the Church-Rosser property. This paper is to show that for the correct construction of parallel programs and for easy reasoning about their correctness the Church-Rosser property is indispensable. A conservative extension of functional systems using a reflection mechanism is proposed for thereduction system π-RED. The resulting reduction system ρ-RED has the Church-Rosser π-RED property. Necessary conditions for reflexive extensions of functional systems are identified. With certain restrictions, the reflection mechanism can also be employed for the integration of different (even non-functional) subsystems, e.g. parsers, numerical packages, and computer algebra systems. Functional programs which have imperative subsystems included by the reflection mechanism can be executed in ρ-RED non-sequentially without causing side-effects due to these subsystems.

Keywords: Reflections, Church-Rosser.

[Zorn *et al.*, 1987]
Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus.
SPUR Lisp: Design and Implementation.
Technical Report UCB/CSD 87/373, Computer Science Division (EECS), University of California, Berkeley, CA, September 1987.

Abstract: This document describes SPUR Lisp, a Common Lisp superset designed and implemented at U.C. Berkeley. Function calling sequences, system data structures, memory management policies, etc. are all described in detail. Reasons for the more important decisions are given. SPUR Lisp is implemented on BARB, a software simulator for SPUR hardware. In addition to describing the design of SPUR Lisp, this paper provides documentation for the BARB simulator, the SPUR Lisp compiler, and associated tools.

Keywords: Lisp, Futures.

[Zorn *et al.*, 1988]
Benjamin Zorn, Paul Hilfinger, Kinson Ho, James Larus, and Luigi Semenzato.
Features for Multiprocessing in SPUR Lisp.
Technical Report UCB/CSD 88/406, Computer Science Division (EECS), University of California, Berkeley, CA, March 1988.

Abstract: This paper describes simple extensions to Common Lisp for concurrent computation on multiprocessors. Functions for process creation, communication, and synchronization are described. Multiple threads of control are created with *process* objects. Communication and synchronization are managed using *mailboxes*. *Signals* provide asynchronous communication between processes. SPUR Lisp includes *future* and *delay* values, which were first introduced in Multilisp. These features provide a flexible and efficient basis on which higher-level multiprocessing abstractions can be implemented and studied.

Keywords: Lisp, Futures.

[Zorn *et al.*, **1989]**
Benjamin Zorn, Kinson Ho, James Larus, Luigi Semenzato, and Paul Hilfinger.
Multiprocessing Extensions in Spur Lisp.
IEEE Software, pages 41–49, July 1989.

Abstract: Spur Lisp is a supreset of Common Lisp that executes on the Spur multiprocessor workstation developed at the University of California at Berkeley. In this article, we present Spur Lisp's multiprocessing extensions.

Keywords: Lisp, Futures.