Chris Hankin, Reinhard Wilhelm (editors)

Functional Languages: Optimization for Parallelism

Dagstuhl-Seminar-Report; 3
3.9.1990-8.9.1990 (9036)

Tel.: +49-6871 - 2458
Fax: +49-6871 - 5942

Das IBFI (Internationales Begegnungs- und Forschungszentrum für Informatik) ist eine gemeinnützige GmbH. Sie veranstaltet regelmäßig wissenschaftliche Seminare,welche nach Antrag der Tagungsleiter und Begutachtung durch das wissenschaftliche Direktorium mit persönlich eingeladenen Gästen durchgeführt werden.

Bezugsadresse: Geschäftsstelle Schloß Dagstuhl
Informatik, Bau 36
Universität des Saarlandes
W - 6600 Saarbrücken
Germany
Tel.: +49 -681 - 302 - 4396
Fax: +49 -681 - 302 - 4397
e-mail: dagstuhl@dag.uni-sb.de

DAGSTUHL SEMINAR
FUNCTIONAL LANGUAGES : OPTIMIZATION FOR PARALLELISM

Organized by :

Chris Hankin (Imperial College, London)
Reinhard Wilhelm (Universität des Saarlandes)

September 3–8 1990

Overview

Chris Hankin and Reinhard Wilhelm

The seminar emphasised four main areas:

- Static analysis and program transformation
- Abstract machines and compilation
- Architectures to support the parallel evaluation of functional languages
- Pragmatics for the control of evaluation order

The majority of the talks concerned the first two topics.

Static analysis and program transformation

Few of the static analysis techniques which have been proposed for functional languages have been concerned with the discovery of opportunities for parallel evaluation or the control of parallel evaluation. The seminar provided a setting for interaction between the theorists working on analysis techniques and implementors. As a result of discussion, a number of analyses were identified as being of interest; these include:

- Detection of Sharing between parallel threads of computation
- Compile-time analysis of lifetimes
- An analysis to remove redundant Sparks in GRIP-like architectures
- Usage analysis
- An analysis to support compile-time load balancing

A more detailed list, which was compiled by Thomas Johnsson, is included in the next section. Another important issue that was raised was how such analyses could be combined.

In contrast, several speakers reported on experiments in program transformation which are directed at better exploitation of parallel hardware. The essence of this approach is a library of higher order functions (e.g. scan) which are suited for implementation on a particular parallel machine model and a transformation algebra for transforming general programs into the required form. This is extremely important work in the light of our concluding remarks below.

Abstract machines and compilation

While there is wide variance between the level of abstraction employed in the different abstract machines, it is possible to identify some emerging trends.

Four of the talks focussed on Term Graph Rewriting, a rewriting formalism invented by Barendregt and co-workers which explicitly captures sharing. The advantage of this approach is that it is readily formalisable, it is an abstraction of the mechanisms involved in many graph reduction machines and so can provide a formal basis for reasoning about the correctness of analysis and transformation techniques.

Certain considerations such as copying of graph structure are not easily describable in Term Graph Rewriting. At a lower level, much of the abstract machine work is based on refinements of the G-machine which has become established as a standard. The most important refinements are ßpinelessnessänd taglessness"; the former involves caching the spine of the graph on the stack so that the whole expression graph does not have to be rebuilt at each step, the latter avoids the need for some indirection by storing code pointers with values rather than type tags.

Architectures

The key discussion between the architects continues to be the fine grain versus coarse grain dispute. The experience of the MIT MONSOON system gives encouraging evidence to support the fine grain approach but offset against this is the early experience with Glasgow's GRIP which requires throttling and a more coarse grain approach. Some of the talks on program transformation suggest that this issue can be hidden from the high-level user. Which class of architecture is "betterïs clearly dependent on the likely job mix.

Pragmatics

Finally, it is becoming apparent that achieving high performance from any parallel architecture will involve significant programmer intervention. The form of intervention may be guidance to a program transformation system or the insertion of control annotations. The need for annotations should come as no surprise given our experience of imperative languages, however the situation is different here since the annotations are semantics-preserving and there is room for optimism that they could be automatically generated by smart static analysis tools.

Acknowledgements

Thanks are due to Josephine Schneider, who ensured that everything ran smoothly at Schloss Dagstuhl, and to Denise Smith who typed most of this report.

Discussion summary: which analysis?

reported by
Thomas Johnsson,
Glasgow University, U.K. / Chalmers University, Göteberg

John Hughes opened the discussion and asked the following question: "Implementors, what analysis would you want from us analysis designers to make your parallel functional implementation run faster?" The following analyses were proposed by the participants:

- Find out which expressions (bits of graphs built) that need locks, since they are shared by parallel processors.
- SPARK elimination: if it can be determined that a certain SPARK is SPARKing something that has been SPARKed before, then eliminate that SPARK.
- Find out which is the cheapest on each case: remote access to data vs. copying the data structure.
- Update elimination: unshared redexes should be identified so that they are not updated (useful also in a sequential implementation).
- Cost estimation (granularity analysis): if the cost of evaluating an expression is sufficiently low, do it on the same processor, thus avoiding the overhead of distributing the computation.
- Time-before-needed-estimation: if a parameter is needed immediately, don't evaluate it on another processor. But if the time before it is needed is long, and the cost of of the parameter is high, do evaluate it on another processor.
- Pipeline buffer size: if you have $f \circ g$, $f : [A] \to B$, $g : C \to [A]$, you have a pipeline. You would like to implement $[A]$ as a buffer rather than using the heap. If g produces and f consumes values at regular intervals you can use a small buffer. But if either f or g needs/produces values in bursts, you need a large buffer. An analysis might help in determining the size.
- An analysis of right hand sides of definitions such that it is known that objects can be delivered unboxed whenever possible for as much data structure as possible. Example:

```
g a b = (a, b)
f a b = a + b
        where (a, b) = g a b
```

I.e., g should return the pair (a, b) simply by putting the two numbers a and b in machine registers.

- Any analysis which enables both
  - a *nice* specification of a sorting algorithm, and
  - sorting such that the updates are performed in situ, i.e., no additional copy is required (the sorting is done in-place).
- An analysis specific to Clean: analysing a functional program (or better, a Clean (TGRS) program) such that the Clean annotations {P} and {I} (which introduce parallelism and control process placement) are put into the right place.

Static Analysis of Term Graph Rewriting Systems

Chris Hankin
Imperial College, London

Term Graph Rewriting (TGR) is the graph rewriting analogue of term rewriting. Its distinctive characteristic is that sharing is explicitly captured. It is intended as an abstraction of the process performed by many graph reduction implementations of functional languages. We are concerned with semantics-based analysis techniques for the optimization of programs represented in this idiom. The talk has three main parts.

Firstly, we review the operational semantics of TGR (presented by Barendregt et al in PARLE'87). The rewriting process is factored into three distinct phases: the build phase during which new nodes are added to the graph, the redirection phase during which the rewrite is effected by switching "pointers" and the garbage collection phase.

In the "classical" approach to abstract interpretation (due to the Cousots) the analysis is factored through a "collecting" semantics. The collecting semantics is the most precise semantics; it collects complete information about program execution and associates it with "program points". Abstract interpretations are presented as abstractions of and proved correct with respect to the collecting semantics. In the second part of the talk we construct a collecting semantics for TGR using the rewrite rules as program points.

Finally, we present an example analysis which performs a form of type inference. The interpretation is essentially similar to the standard semantics but uses abstract variants of the three phases of rewriting. We show how the correctness of this analysis can be established using adjoined functions in the "classical" style. We suggest that we have established a general framework for such analysis.

Extending Compile Time Garbage Collection to Parallel Implementations

Simon B Jones
University of Stirling, Scotland

One of the most significant factors influencing the performance of functional programs is their uneconomical use of dynamic data storage: functional semantics prevents the programmer from explicitly indicating that storage can be reused, and thus expensive

runtime heap management software is usually required (mark/sweep, stop-and-copy, reference counting, etc). It seems likely that a functional program's performance can be improved if we optimize it, at compile time, to include explicit unconditional storage de-allocation or re-allocation operations rather than code for runtime decision making (e.g. replacing a graph marking scan or a decrement and test of a reference count). Hudak and Bloss have investigated static program analysis techniques for updating atomic or aggregate data structures "in place": abstract interpretation is used to locate operations in a program which always consume the last reference to an operand, and which immediately create a new object of identified structure. Le Métayer and the author have investigated the application of a similar technique to a strict language with list structures, and the result of this in the sequential case have been published in the proceedings of FPCA'89. It is important that we consider extending the technique to the context of the parallel evaluation of functional programs; this work has not started yet, and the author presented some preliminary thoughts to the workshop.

In the sequential context the approach is straightforward, though the details are complex: we proceed from a standard semantics of our functional language, to a standard storage semantics which includes a heap management model, and then to a non-standard form of the semantics which controls storage allocation and de-allocation via sharing and "future use" information obtained by abstract interpretation (the storage cells that can be collected during evaluation of an expression are those that are not shared with previously evaluated expressions, and are not needed by expressions still be evaluated).

How must this approach be adapted to the parallel case? It would appear that the standard storage semantics, being an operational model, will need to be extended to a parallelism model; this is not a simple problem. Similarly, perhaps the non-standard semantics expressing the desired optimizations will need to be a parallelism model. However, let us consider a simple implementation of parallelism in which the individual arguments in an argument list are evaluated in parallel and then synchronization occurs before the function in entered. In this case the desired optimization can be described without explicitly modelling parallelism: we must compile every argument in an argument list under the assumption that all the other arguments are still to be evaluated, since this is the worst cast that can occur with unpredictable, interleaved, parallel evaluation of the argument list. This storage allocation and deallocation can be controlled by "future use" information, and it appears that the sharing information may be redundant.

Analysis of Functional Programs by Abstraction of Pre-Postconditions

Torben Æ. Mogensen
DIKU, Denmark

We investigate the idea of using the notion of pre- and post conditions to make an automatic analysis of functional languages.

We start by defining our concept of a condition: a continuous function from a value domain into the domain of booleans.

Conditions are partially ordered by implication, and pre/post-conditions are defined in terms of the ordering.

We then identify a condition by the pair of inverse images of true and false, and restate the definitions of implication and pre/postconditions in terms of such pairs of sets. We find that we can compute the weakest precondition and strongest postcondition if we can find the images and inverse images of functions applied to the sets from a pair.

We then present a small functional language and its semantics and show how images and inverse images of functions defined in that language can be found. The method is, however, not computable, so approximations should be used.

Derivation of parallel scan

John T. O'Donnell
Glasgow University

Mapping algorithms onto parallel architectures can be a difficult problem. Equational reasoning and hardware modelling in a functional language provide powerful tools for solving this problem. An example of the method is the derivation of a parallel implementation of scan for an abstract tree machine.

We begin by specifying the scan algorithm as a functional program. (Scan is an important algorithm with many practical applications.) Trying a divide and conquer approach, we derive a scan decomposition law which shows how a large scan can be defined by combining two smaller scans. This law formalizes the intuition behind a parallel implementation of scan.

Next we give a formal specification of the parallel architecture; this takes the form of a function m, such that $(m\ p\ x)$ is the result of running machine m with program p on data x.

The crucial step is to formally state a conjecture that there exists a program p which causes machine m to execute the algorithm a. This conjecture takes the form

$$\exists p.\forall x.m\ p\ x\ =\ a\ x$$

Let m be the abstract tree architecture, let a be the function which computes scan and fold, and let x be an arbitrary argument to scan. The only unknown value is p, and equational reasoning suffices to solve the equation for p. This derivation achieves three goals:

1. it proves that a tree machine can execute scan in parallel,
2. it gives us the parallel program p, and
3. it provides a correctness proof for p.

The result turns out to be the fastest known algorithm for scan. The algorithm was originally derived using ad hoc methods, and apparently it has never before been proved correct.

This work relies on the following properties of functional languages: referential transparency, nonstrict semantics, higher order functions, polymorphic types and algebraic types. It provides supporting evidence for the thesis that formal reasoning about functional programs can help to solve practical problems.

Analyzing and Transforming Functional Programs

Hanne Riis Nielson
Aarhus University, Denmark

It is often useful to regard the efficient implementation of functional programs as arising from two stages. The post-processing stage performs a rather naive implementation whereas the pre-processing stage achieves a better overall result by performing program transformations.

We shall first discuss this paradigm in the context of binding time analysis. The binding time analysis will ensure a clear separation between the binding times of interest.

However, in doing so, it will often defer too many computations to the base binding time. To avoid this it is helpful to re-arrange the program before performing the binding time analysis. We show how the well-formedness rules for the binding times may be used to derive a disagreement point analysis that explicitly indicates where it will be beneficial to apply program transformations that change the overall type of the program. We sketch how the approach may be extended to recognize sub-tasks that may be performed in parallel on different processors.

In general, program transformations need to exploit the results of abstract interpretation. In the second part of this talk we take a first step in that direction by restating Phil Wadler's strictness analysis for lists in our framework of parameterized semantics. We succeed in obtaining a composite description of his case analysis using two techniques. One is to approximate the Booleans by a four-point diamond domain rather than the usual two-point chain domain. The other is to use a kind of inverse cons operation that maps into a tensor product rather than a cartesian product. We conclude by discussing the benefits of the parameterized semantics approach to specifying analyses.

On Parallel Evaluation of Expressions in Function Languages

Hugh Glaser
University of Southampton, UK

Almost since the first introduction of subroutines in computer languages, there has been widespread discussion of the varieties of evaluation mechanisms for parameters. The introduction of new and different models from the traditional von Neumann approach, and more recently the work on parallel execution, has meant that there are now a large number of terms available to describe the parameter passing mechanisms. Unfortunately these terms do not describe the complete set of options open to the language implementor and machine designer and in addition it is now being recognised that the complexities of the new parallel machines require the ability to describe more complex evaluation mechanisms. In this paper we look at the situation for functional languages, focusing on the more general idea of expression evaluation time, part of which is the parameter evaluation mechanism, and discuss the variety of options that are available.

Mapping Functional Programs onto Parallel Machines

John Darlington
Imperial College, U.K.

We address the problem of how applications can be implemented efficiently on existing parallel machines without compromising desirable software characteristics such as comprehensibility, modifiability and portability. Current experience indicates that explicit control over a machine's resources is necessary for the efficient exploitation of the available parallelism but if this is achieved through the use of imperative programming languages the programmer's task becomes very complex.

Our approach is to use functional programming languages to model general parallel computation and identify the characteristics of particular machines with subsets of general program forms and to use program transformation to convert general programs into a form that can be efficiently implemented on the target machine.

The target program forms are represented by a range of program skeletons or higher order functions representing useful algorithms paradigms that can be efficiently implemented on particular parallel machines. Skeletons have been produced, so far, for pipeline, mesh, co-operating specialists, divide and conquer and process farms and a range of transformation algebras have been developed for each particular skeleton. The skeletons can be implemented sequentially or as specially optimized functions on particular parallel machines.

Improving Graph Reductions Code by In-line Expansions of EVALS

Thomas Johnsson
Glasgow University, U.K. / Chalmers University, Göteberg

In imperative programs for efficiency the bulk of the work is done in relatively big procedures, and calls and returns are relatively rare. In such situations there are well-established techniques for generating good machine code, making good use of machine resources, the registers in particular.

On the other hand, a typical functional program consists of a large number of small functions. Therefore, the code from functional programs have a much higher call/return overhead.

A common 'trick' in dealing with this overhead is to make function bodies bigger by doing inline expansion of functional calls. But this is straight forward only if the applied functional has a 'known' body (this rules out functions passed as arguments). An additional difficulty with this is caused by laziness: in addition to the calls visible in the source program, the intermediate graph reduction code for lazy programs also contain a lot of calls to EVAL, making it nearly impossible to use machine registers.

In this work, the intermediate language (called PG-code) is essentially a procedural version of G-machine code with the G-machine instructions as 'three address code'. An essential feature is that in the PG-code from a functional program, EVAL is an 'ordinary' procedure. It is thus possible to do in-line expansions also of EVAL calls.

Once an in-line expansion of an EVAL call has been made, other improving transformations present themselves. For instance, we are able to do a transformation to obtain an analog to what the spineless tagless G-machine does at runtime with its vectored return mechanism.

Compiling a Functional Language for Fine-grained Parallelism

Rishiyur S. Nikhil
MIT Laboratory for Computer Science

Fine-grained parallelism is useful in large scale MIMD machines in order to tolerate long inter-node latencies. If a program is partitioned into many, small threads, each processor can have an adequate pool of threads to keep it busy even though several threads may be suspended on inter-node communications.

We are building a compiler to translate non-strict functional languages into fine-grained threads. It has three interesting intermediate languages: a Flat First Order Language, Dataflow Graphs and P-RISC Abstract Machine Code.

After conventional front-end processing (type checking, lambda lifting, case compilation, etc.) we translate the resulting supercombinators into a Flat First-Order Language (FFOL), in which all higher order functions and applications have been coded into explicit closure manipulation operations, with closure representations being chosen on a case-by-case basis. FFOL expressions are constants, variables, Letrecs, conditionals (case) and simple first order applications.

FFOL code is translated into dataflow graphs (DFG+s) which depict all the explicit data dependencies as control flow arcs. Implicit data dependencies (through data structures) are handled by assuming operators for synchronized access to heap locations (I-store and I-fetch).

The DFG is translated into code for the P-RISC abstract machine, which makes explicit the threads in the DFG and their synchronization, the frame locations needed by a procedure, and the procedure calling conventions. P-RISC code is close to conventional RISC code.

Finally, P-RISC code is translated into native code for various existing machines and for some new, multi-threaded architectures.

A prototype of the compiler has been implemented, up to the generation of P-RISC code. We hope to complete a P-RISC to C back end in the next few months, and to improve the optimizations at each stage. After this, we can begin performance evaluations and experiments in resource management.

Provably Correct Compilation of Functional Programs

Flemming Nielson
Aarhus University, Denmark

Functional languages like Miranda and Haskell employ a non-strict semantics. This is important for the functional programming style as it allows one to compute with infinite data structures. However, a straight-forward implementation of the language will result in a rather inefficient implementation. In this talk we consider the compilation from a large (categorical) combinator notation to a version of the categorical abstract machine.

In the first part we consider ways of reducing the number of DELAY and RESUME instructions generated in the naive approach. One scheme exploits the use of local strictness information and another the use of strictness information as right context. Together they allow to avoid generating a good part of the superfluous DELAY instructions. Combined with a notion of evaluation degrees as left context this allows also to avoid generating a good part of the superfluous RESUME instructions. For the factorial program they result in the same code as would have been produced in an eager scheme.

In the second part we consider the correctness of the compilation. Even the well-behavedness of the code generated in the naive approach is surprisingly hard to ensure. We show how the techniques of Kripke-like relations indexed by a two-level type structure may be used to overcome this.

Ultimate GOTO Considered Interesting (Work in Progress)

Lennart Augustsson
Chalmers University, Sweden

In constructive type theory you write programs by first writing the specification in a (kind of) predicate logic. The next step is to prove that the specification, viewed as a logical proposition, is true. Since each proof rule corresponds to a program construct, it is then possible to extract a program from this proof. This corresponds to the other view of the logic: viewing propositions as sets and programs as elements in the sets.

There are an (infinite) number of propositions that can be proved by classical logic, but not by constructive logic. The most well known is the law of excluded middle, $A \vee \neg A$ (as usual $\neg A \equiv A \rightarrow \emptyset$, where $\emptyset$ is the false proposition or the empty set). The proposition $((A \rightarrow B) \rightarrow A) \rightarrow A$ is called Pierce's axiom, and it is valid classically but not constructively (using it the law of excluded middle can be proved and *vice versa*).

The language Scheme has a primitive function `call/cc`. This may be considered as the ultimate goto. `Call/cc` sets up a 'label' in the computation to which later computations may "jump". The continuation style denotational semantics for `call/cc` is very simple, it just takes the current continuation and gives it to its argument. The type of `call/cc` is $((A \rightarrow B) \rightarrow A) \rightarrow A$.

This suggests a connection between Pierce's axiom, i.e. classical logic, and `call/cc`. Using `call/cc` as the realizer for Pierce's axiom it is possible to extract programs from classical proofs. A number of classical propositions give rise to quite reasonable programs. E.g. $(A \rightarrow (B \vee C)) \rightarrow ((A \rightarrow B) \vee C)$, which gives a kind of error handler.

Using `call/cc` will not really give any new expressive power because it is possible to run a program with `call/cc` in a language without `call/cc` by writing an interpreter. The advantage is that the program with `call/cc` is simpler.

Continuation-Based Parallel Implementation of Functional Languages

J-F Giorgi and D. le Métayer
IRISA/INRIA, Rennes, France

Using the concept of continuation, we have designed a sequential compiler for functional languages based on successive program transformations. The output of the compiler, while still in a functional language, can be seen as traditional machine code. Continuations are used to model the computation rule of the language (a simple λ-calculus with constants and call-by-value). We propose here an extension of the role of continuations for a parallel implementation; all control transfers are continuations, exportable tasks are continuations, exported tasks are replaced in the stack by continuations which are executed if the processor reaches the continuation before its result is returned, the task of sending the result to the original processor is the continuation of the exported task. As a result of this policy, there is no task management system in our implementation and the code produced for an inherently sequential program is exactly the code produced by the sequential version of the compiler and almost no overhead is incurred for the creation of tasks. The first results are promising but a full-scale implementation of a real language with complex data structures and call-by-need is necessary to fully assess this approach.

Work in Progress on Compiling Caliban

Paul Kelly
Imperial College, U.K.

The aim of this work is to develop powerful, simple tools to aid programmers in exploiting the capabilities of existing parallel processors. The goal is to give the programmer enough control over the machine to get the performance of which it is capable, while providing the maximum level of assistance with abstraction mechanisms in the programming language.

This talk concerns Caliban, a declarative annotation language which augments a pure functional language. Caliban annotations in their simplest form resemble Peyton Jones' "Spark"s, or Halsteads "Futures". The difference is that in Caliban annotations are collected in a declarative description of the "processor network" to be created. The annotations take the form of assertions about placement and communications, and these assertions can be generated by user-defined functions. This allows networks to be parameterized, and enables recurring structures to be captured using "network forming operators".

The work reported in this talk is aimed at the simple case where a program has a static process network. This occurs just when the program can be simplified by partial evaluation to a basic "normal" sublanguage. In this case, a network of communicating functional processes (in fact a Kahn network), can be derived. Unfortunately, this

process does not easily lead directly to sequential Kahn processes. The problem occurs, for example, when a processor has two output streams: there is no sequential reduction order for the processor which both respects termination semantics, and exploits vertical parallelism. We are thus forced either to use a parallel reduction strategy, with two fairly-scheduled reduction processes, or to avoid pre-computing a value of either output stream until a demand token is received determining which is needed.

An alternative view is that the programmer should control this behaviour explicitly, for example by rewriting the program so that a stream of pairs is communicated rather than a pair of streams.

Binding-time Improvement for Free!

Carsten-Kehler Holst and John Hughes
University of Glasgow

Any curried function may contain expressions that depend only on the earlier parameters. When such a function is partially applied, the evaluation of these static expressions may be shared between all calls of the partial application. Fully lazy implementations and partial evaluators take advantage of this possibility to realise significant performance improvements.

Unfortunately, staticness is a delicate property. In practice, programs intended for partial evaluation must be carefully tweaked to improve their binding-times. This process is tricky, error-prone, and tedious.

We are interested in transformations that improve binding-times. One useful class of transformations is the "commutative" laws, which enable static operations to be brought closer to static data, thus enlarging the static parts of the computation. But there are an unlimited number of such laws.

We have examined the possibility of using only laws that follow from the polymorphic types of functions, à la Wadler's "Theorems for Free!". In the case of first order functions, the "free theorem" is just a commutative law. For higher-order functions, the free theorem is a conditional commutative law, which can be used for transformation once the conditions are solved for function-valued unknowns. We have defined a pre-order on functions which allows these conditions to be expressed as a number of lower bounds for the unknowns; these can be solved by taking lubs.

We have applied these techniques to a number of examples, including transforming an interpreter into a compiler. These examples show that free theorems suffice to make significant binding-time improvements. An automated binding-time improver based on this work seems feasible, and would be a useful programming tool.

The Implementation of Functional Logic Languages

Hendrik C.R. Lock
GMD National Research Laboratory, University of Karlsruhe

Functional Logic Languages combine the features of the two main declarative programming paradigms. A whole bunch of such languages has been proposed over the recent years. On one side of the spectrum we find Horn logic languages extended by functional features, and on the other end generalizations of functional languages by unification and non-determinism. Their operational models involve deterministic and ambiguous term rewriting, narrowing and SLD resolution.

In order to characterize different classes of such languages by their operational semantics, a calculus was presented which consists of a syntax $\mathcal{L}_0$ and a reduction semantics. This base calculus is an extension of the λ-calculus by: 1st order terms, logical variables, 1st order unification (which subsumes pattern matching), a choice operator, a guard operator and conjunction. In particular, its reduction semantics preserves sharing wrt. substitution. Then, a language class is defined by some (abstract) syntax $\mathcal{L}_x$ and by a translation of $\mathcal{L}_x$ into $\mathcal{L}_0$.

Furthermore, a general implementation technique was presented which supports each of the features contained in the base calculus including all of their combinations. It consists of a design space of abstract machines, each of them supporting a particular combination. Thus, a direct correspondence is obtained between the language classes and abstract machines implementing them.

The design space consists of a core machine and orthogonal extensions. The core is derived by unifying the common principles of functional and logic machines, and it turns out to be the "classical" ALGOL 60 technology underlying the implementations of procedural languages. Accordingly, the core supports all features of first order functional languages. The four extensions respectively implement *lazyness, higher order functions, unification,* and *backtracking.*

An instance of this design principle is the JUMP-machine which integrates the core and all extensions. It implements the class based on combinations of reduction, unification and SLD resolution. It also has been shown how another simple extension suffices for correct and complete implementations of *lazy narrowing.*

The careful design of the JUMP machine shows that the logical support does not introduce run-time overhead whenever ground term reduction is performed. In this case we can expect that the efficiency of functional machines such as the "Spineless

Tagless G-Machine" can be preserved. The effect on memory consumption is not yet known, at least it is clear that deallocation becomes much more involved. The logical parts of the machine are efficiently designed by following the principles of the "Warren Abstract Machine".

Distributed Applicative Arrays

Herbert Kuchen
RWTH Aachen, Germany

Lists, the typical data structure of functional languages, force a sequential treatment of their elements and are hence badly suited for parallel implementations. Some other applicative data structures are proposed, which are appropriate for implementations of such languages on loosely coupled multiprocessor systems. Besides so called sequences, i.e. list-like structures internally implemented by binary trees, we mainly consider distributed applicative arrays (DAA's).

A DAA is distributed among the stores of the processing units. Each element is accessed via its virtual address. Each processing unit maintains a table which translates the virtual address of the accessor is stored and later on used to transmit a copy of the element, when it is ready.

For functions like map, fold, and zip, DAA's have the advantage that the number of messages only depends on the number of processing elements, not on the number of DAA elements. This is not the case if lists or sequences are used.

For some example programs DAA's were between 4 and 16 times faster than sequences. The experiments were performed on a system with 1, 12, 48 and 64 processors respectively. The implementation is based on the parallel abstract machine PAM, which uses programmed graph reduction.

Extending a Graph Reduction Machine for the Implementation of a Functional Logic Language

Rita Loogen
RWTH, Aachen

The talk presents joint work with Herbert Kuchen (RWTH Aachen), Juan Jose Moreno-Navarro (Madrid) and Mario Rodríguez Artalejo (Madrid).

During the last years, several approaches have been proposed to achieve an integration of functional and logic programming languages in order to combine the advantages of the two main declarative programming paradigms in a single framework. The so-called functional logic languages retain functional syntax but use narrowing—an evaluation mechanism that uses unification instead of pattern matching for parameter passing—as operational semantics.

We present an implementation of the higher-order lazy functional logic language BABEL on (the sequential kernel of) a parallel graph reduction machine that has been extended by the logic features, namely unification and backtracking.

Lazy evaluation is supported by an automatic transformation that eliminates non-flat sub-unifiable program rules. The resulting program allows an easy determination of demanded arguments of a function symbol f, because in either all or none of the program rules for the formal parameter is a non-variable term.

Finally, we discuss the possibilities to exploit implicit parallelism in the functional logic framework.

Some Early Experiments on GRIP

Kevin Hammond and Simon Peyton-Jones
Glasgow University, U.K.

GRIP is a multiprocessor designed to execute functional programs in parallel using graph reduction. We have implemented a compiler for GRIP, based on the Spineless Tagless G-Machine, and can now run parallel functional programs with substantial absolute speedup over the same program running on a uniprocessor Sun.

Parallel functional programming shifts some of the burden of resource allocation from the programmer to the system. Examples of such decisions include when to create a new concurrent activity (or thread), when to execute such threads, where to execute them, and so on. It is clearly desirable that the system should take such decisions, provided it does a good enough job. The big question for parallel functional programming is whether good resources allocation strategies exist, and how well they perform under a variety of conditions.

Now that we have an operational system, we are starting to carry out experiments to develop resource-allocation strategies, and measure their effectiveness. This talk reported on some very preliminary results, mainly concerning the issue of when, or even whether, to create a new thread. This is an aspect which has so far received little attention—other work has focussed mainly on load sharing rather than thread creation.

Our results confirm the importance of effective throttling strategies to limit parallelism, especially ones capable of adapting dynamically to the characteristics of a particular program. Simple strategies give useful improvements, but much work is needed to refine these strategies. as we make improvements based on the statistics we have gathered, so the performance of normal functional programs should improve.

Probabilistic Load Balancing for Parallel Graph Reduction

Helmut Seidl and Reinhard Wilhelm
Universität des Saarlandes, Germany.

We analyze simple probabilistic implementations of (slightly restricted) parallel graph rewriting both on a shared memory architecture like a PRAM and a more realistic distributed memory architecture like a transputer network.

Graph rewriting is executed in cycles where every cycle consists in the execution of all the tasks presently available in the graph. Assume there are p processors and N executable tasks in the cycle. We are able to show: the PRAM can execute the cycle in (optimal) time $O(\frac{N}{p})$ with high probability provided $N = \Omega(p^2 \log p)$, whereas a processor net can execute the cycle in time $O(\frac{N}{p} \log p)$ with high probability using chunks of messages of size $O(\frac{N}{p})$ if only $N = \Omega(p \log p)$.

Implementation of A Parallel Functional Language

Martin Raber
Universität Saarbrücken

Our approach to the implementation of a parallel functional language is shown. It is based on a parallel abstract machine which is a straightforward parallelization of Johnsson's G-machine modified due to some observations Fairbairn and Wray made in their Tim article.

The presentation is divided into three parts :

- The parallel functional language and its compilation to machine code.
- Some features of the abstract machine.

- The realization of that machine on a transputer network.

A Pragmatic Approach to the Analysis and Compilation of Lazy Functional Languages

Hugh Glaser, Pieter Hartel and John Wild
University of Southampton, U.K.

The aim of the FAST Project is to provide an implementation of the functional language Haskell on a transputer array. An important component of the system is a highly optimizing compiler for Haskell to a single transputer. This talk presents a methodology for describing the optimizations and code generation for such a compiler, which allows the exploitation of many standard and some new techniques in a clear and concise notation. Results are included showing that the optimizations give significant improvement over the standard combinator and (Johnsson's 1984) G-machine implementations.

Compiling Functional Languages Based On Graph Rewriting

John Glauert
University of East Anglia, Norwich, U.K.

We extend techniques of Kennaway (TCS'90) to allow a general functional program expressed as a Term Graph Rewriting System to be transformed to a much simpler TGRS which may be converted to machine code directly. Programs may also be executed in the practical graph rewriting language, Dactl.

The key technique is to separate pattern matching of rules from evaluation of arguments, ensuring that arguments are sufficiently evaluated before a rule is invoked. Information about the degree of evaluation enables optimization to be made so that the resulting rules have a dataflow style.

We use a rewriting framework in order to allow programs to be reasoned about and transformed. First-order functions are handled, but rules to 'flatten' higher-order programs exist.

Concurrent Functional Programming

Rinus Plasmeijer and Marko van Eekelen
University of Nijmegen, The Netherlands

The primitives for process creation in the functional language Concurrent Clean already enable the specification of all kinds of process behaviour. However, for an average programmer it is difficult to get a clear view of the process structure being defined. Higher level primitives with a restricted power are needed to allow functional programmers to define relatively simple kinds of dynamically changeable process behaviour. Two such primitives are proposed with which the potential power of concurrent functional programming is demonstrated. In a concurrent functional language processes are functions that are executed concurrently. By using mutual recursion arbitrary dependencies between these functions can be specified thus creating a way to define arbitrary, possibly cyclic, process networks. The communication between the processes is defined implicitly and it is driven by the lazy evaluation order. No extra primitives for communication are needed: communication takes place when a process demands a value that is being calculated by another process. An important aspect of the introduced primitives is that in contrast with the primitives of Concurrent Clean they force evaluation of the indicated expressions to normal form instead of the root normal form (a weak head normal form). The user has to take care with help of a strictness analyzer that the semantics are not changed.

Concurrent Clean—Status Report

Rinus Plasmeijer, Marko van Eekelen, Erik Nocker and Sjaah Smetsers
University of Nijmegen, The Netherlands

Concurrent Clean is a lazy, high-order functional language based on Term Graph Rewriting Systems. Clean has an explicit notion of sharing and copying graph structures. The language includes Modula2-like modules and a Milner-Mycroft type system with algebraic, synonym and abstract types.

The language is designed to let the programmer explicitly control the reduction order via annotations. Lazy evaluation can be turned into eager evaluation, (partially) strict datatypes can be specified. Parallel evaluation can be defined using the concept of lazy copying in the semantics. Two annotations to spark off processes are employed: one with which a parallel process can be created on another processor, one with which

an interleaved process can be created on the same processor. For the communication between parallel executing processes a copy of a graph structure is made in such a way that the indicated graph is copied upto the nodes where other processes are executing. Lazy copying makes it possible to choose between shipping data or shipping work. Dynamically changeable arbitrary process networks can be specified.

The Concurrent Clean system includes a powerful and fast strictness analyzer based on abstract reduction. Clean is compiled to abstract machine code (PABC machine). A simulator is available that simulates the parallel behaviour (runs on Mac, Sun, Atari, PC). A code generator for MacII and Sun3 is available that generates very good code: nfib 303,000 calls per second on MacIIx (MPW-C runs at 180,000 calls per sec.). Fast Fourier of 8×1024 elements cost 16 seconds + 19 seconds garbage collection (The corresponding imperatively-written C program takes 8 seconds). So, the performance of Concurrent Clean programs is becoming comparable with C.

Bisher erschienene Titel:

W. Gentzsch, W.J. Paul:

Architecture and Performance, Dagstuhl-Seminar-Report 1 (9025), 18.6.1990 - 20.6.1990

K. Harbusch, W. Wahlster:

Tree Adjoining Grammars, 1st. International Worshop on TAGs: Formal Theory and Application, Dagstuhl-Seminar-Report 2 (9033), 15.8.1990 - 17.8.1990

Ch. Hankin, R. Wilhelm:

Functional Languages: Optimization for Parallelism, Dagstuhl-Seminar-Report 3 (9036), 3.9.1990 - 7.9.1990

H. Alt, E. Welzl:

Algorithmic Geometry, Dagstuhl-Seminar-Report 4 (9041), 8.10.1990 - 12.10.1990