Werner Damm, Chris Hankin,
John Hughes (editors):

Functional Languages: Compiler Technology and Parallelism

Dagstuhl-Seminar-Report; 36
23.-27.3.92 (9213)

ISSN 0940-1121

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

Das Internationale Begegnungs- und Forschungszentrum für Informatik (IBFI) 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: office@dag.uni-sb.de

Dagstuhl-Seminar

Functional Languages: Compiler Technology and Parallelism

Organized by:

Werner Damm (Universität Oldenburg)

Chris Hankin (Imperial College, London)

John Hughes (University of Glasgow)

March 23-27, 1992

Overview

Werner Damm, Chris Hankin and John Hughes

The seminar emphasized four issues:

- static program analysis
- extensions for programmer control of parallelism
- functional+logic languages and constraints
- implementation of functional languages

There were two formal discussion sessions, which addressed the problems of input/output in functional languages, and the utility of static program analysis.

It is gratifying that the first Dagstuhl seminar in this area (Functional Languages: Optimization for Parallelism) had stimulated many developments which were reported at this one.

A particular feature of this seminar was the large number of prototypes which were demonstrated and which vividly illustrated the issues raised in discussions and presentations.

Static Program Analysis

Static program analysis has been thoroughly investigated for optimising sequential implementations, but parallel ones offer new problems. Discovering properties of synchronisation, for example, requires richer domains than those used in the sequential setting, leading to a combinatorial explosion in cost. Current sequential analyses operate at or beyond the limits of today's algorithm technology. The most expensive aspect is computing fixpoints, which requires a convergence test and therefore a decision procedure for equality of abstract values. New work reported here helps reduce the need for convergence tests.

Much progress has been made in vectorising imperative programs, and it is important that we capitalise on this. At this seminar we saw how to apply these techniques to the parallelisation of array expressions.

Extensions for Programmer Control of Parallelism

There are two approaches to parallel functional programming: the *implicit* approach requires the compiler to identify suitable opportunities for parallelism, while the *explicit* approach places this responsibility on the programmer. In discussions a strong feeling emerged that the implicit approach is intractable at present, which motivates an active interest in the explicit alternative.

One way to add explicit parallelism is to integrate functional languages with process notations such as CCS. We heard of two efforts in this direction, one focussing on congruence properties and algebraic laws, and the other on type systems that can represent communication. But this approach risks compromising referential transparency.

Quite a different direction is to add new datatypes that support parallelism, such as the multiset. The Gamma formalism can express multiset programs with a minimum of implied sequencing. Multiset programs risk indeterminacy, but new work on confluence in Gamma promises to control this risk.

Functional+Logic Languages and Constraints

The integration of functional and logic programming aims at a well-founded combination of two important declarative programming paradigms in a single framework. The main research directions in this field are language design and implementation. Some efforts in language design were reported on a further integration of functional+logic languages and constraints. Implementations of functional+logic languages are based on the implementing techniques for logic programming such as WAM (Warren Abstract Machine) and functional languages such as graph reduction. Both sequential and parallel implementation techniques were presented. An interesting issue was raised: how to

apply and combine the existing functional and logic static analysis in a functional+logic framework.

Implementation of Functional Languages

A perennial issue in the implementation of functional languages is input/output. Early implementations offered only weak and inefficient I/O interfaces. At this seminar two new approaches were presented, both enabling functional programs to perform arbitrary I/O operations with the same efficiency as any other program. Both however force the sequentialisation of I/O, which may be inconvenient and, in a parallel context, inappropriate.

In the two years since the first seminar commercial parallel machines have become much more common, and there has been a corresponding shift away from designing new architectures towards exploiting stock hardware. Good sequential technology is a prerequisite for good parallel implementations, and both sequential and parallel implementation work was well represented.

An exciting new development at this seminar was the emergence of profiling and monitoring tools as an essential component of the programmer's workbench. Historically the very-high-level nature of functional programs has made it difficult to understand their space and time behaviour. A truly remarkable result presented at the seminar showed that, using profiling tools, a few small modifications led to a 440-fold improvement in the cost of a medium-sized program.

Acknowledgements

Thanks are due to the staff at Schloß Dagstuhl, who ensured that everything ran smoothly, and Feixiong Liu for coordinating the production of this report.

Programme of the Workshop

Monday, March 23

Session 1 : I/O Functional Languages

8.45 - 9.45 **Rinus Plasmeijer:** I/O in functional languages

9.45 - 10.45 **Kevin Hammond:** Monads and I/O

Session 2 : Implementation of Functional Languages I

11.15 - 12.15 **David Walkeling:** Heap profiling of lazy functional languages

14.00 - 14.30 **Denis Howe:** Implementation of evaluation-transformers

14.30 - 15.30 **Guido Hogen:** Automatic parallelization of lazy functional programs

Session 3 : Implementation of Functional Languages II

16.00 - 16.30 **Lennart Augustsson:** Splitting infinite sets of unique names by hidden state chabges

16.30 - 17.15 **Peter Wentworth:** Code generation for a lazy functional language

17.15 - 18.00 Discussion session: I/O in functional languages

Tuesday, March 24

Session 4 : Abstract Interpretations

9.00 - 9.45 **Fritz Henglein:** Straight-line code generation and evaluation by partial evaluation

9.45 - 10.45 **John Hughes:** Abstract interpretation of parallel functional programs

11.15 - 12.15 **Hanne Nielson:** Computing fixed points in static program analysis

Session 5: Compile-time Optimisations

14.00 - 15.00 **Paul Kelly:** Dependency analysis in array computations

15.30 - 16.30 **Manuel Chakravarty:** The structure of a back-end to compile functional languages

16.30 - 17.15 **Martin Raber:** Small functions should be evaluated

17.15 - 18.00 Demonstrations: I/O in concurrent clean by **Rinus Plasmeijer** and I/O with Monads by **Kevin Hammond**

Wednesday, March 25

Session 6 : Logic + Functional Languages and Constraints

8.30 - 9.20 **Peter van Roy:** Constraints + control = compilations; or How to compile matching and resolution in LIFE

9.20 - 10.10 **Feixiong Liu:** An OR-parallel implementation of K-LEAF based on graph rewriting

10.40 - 11.25 **Francisco Lopez-Fraguas:** A scheme for constraint functionl logic programming

11.25 - 12.15 **Herbert Kuchen:** Implementing a lazy functional-logic language with disequality constraints

Thursday, March 26

Session 7 : Lambda Calculus and Processes I

9.00 - 9.45 FACILE: **Bent Thomsen:** A functional language for parallel programming

9.45 - 10.30 **Lone Leth Thomsen:** A new direction in functions as processes

11.00 - 12.00 **John Glauert:** Process networks and graph rewriting

14.00 - 15.00 **Flemming Nielson** TPL : the typed λ-calculus with first-class processes

Session 8 Lambda Calculus and Processes II

15.30 - 16.30 **Daniel le Metayer:** Programming by multiset transformation

16.30 -17.15 **Chris Hankin** Confluence properties of gamma-programs

17.15 - 18.00 Demonstrations : LIFE by **Peter Van Roy** and Haskell-style by **Lennart Augustsson**

Friday, March 27

Session 9 : Implementation of Functional Languages III

9.00 - 9.45 **Heiko Dörr:** Graph-Grammar based monitoring of parallel graph reduction machines

9.45 - 10.45 **David Lester:** Garbage collection

11.15 -12.15 Discussion Session: Is Automatic program analysis of any use ?

I/O in Functional Languages

Rinus Plasmeijer
Univeristy of Nijmegen

The specification of I/O always has been one of the weakest points of functional languages. I/O needs side-effects and precise order of evaluation which is not easy to realize. In the talk a new method for the handling and specification of I/O is presented based on an environment passing scheme and a structured, object oriented programming style. This combination yields a high level and elegant functional specification method for I/O, called event I/O. An I/O system based on this method has been implemented in the lazy functional graph rewriting language Concurrent Clean, developed at the University of Nijmegen. The results are very encouraging: interactive programs written in Concurrent Clean are concise, easy to write and comprehend as well as efficient. The presented solution can be applied for any other functional language as well.

Monads and I/O

The GRASP team, Glasgow University.

Taking the concept of monads from category theory (as espoused by Moggi or Wadler), we have developed a structure for general procedure call within a pure functional language. Our model allows almost any imperative function to be called from a pure functional language *without* loss of referential transparency.

The key to our approach is to view imperative procedures as actions operating on an implicit state. Actions are sequenced by the monadic structure of our program, and are full first-class citizens: they may be passed to functions, returned as the result of a function, included in data structures etc. exactly as other values in a functional program.

```
data C a = C (a,State); data State = State
unitC :: a → C a
unitC a = C (a,State)
bindIO :: C a → (a → C b) → C b
x `bindC` k =
        \s → case x s of
                C (a,s') → k a s'
```

Although this system is strictly no more powerful than the Hope+ continuation-style, a major contribution of our work is to make it possible to program arbitrary IO and other operations entirely in a functional language. Unlike other approaches, there is no magic "wrapper" which interprets IO requests. Consequently, imperative calls are entirely open to our functional transformation system, and may be freely manipulated using higher-order functions. For example, the Haskell

```
copyfile :: File → File → C ()
copyfile f1 f2 =
            getc f1          `bindC` (\ c →
            if c == eof then
                unitC ()
            else putc c f2           `bindC` (\_ →
                copyfile f1 f2))
```

can be compiled into C code which approximates the following:

```
{ int c;
  copy: c = getc(f1);
  if c == -1 then
   return;
  else {
   putc(c,f2);
   goto copy;
```

}
}

Heap Profiling of Lazy Functional Programs

Colin Runciman and David Wakeling
University of York

The *extensional properties* of a functional program (what it computes as its result) are usually far easier to understand than those of the corresponding imperative one. However, the *intensional properties* of a functional program (how it computes its result) can often be much harder to understand than those of an imperative one, especially in the presence of higher order functions and lazy evaluation. We have developed a tool for profiling the space consumption of lazy functional programs by *heap profiling*. This talk describes some aspects of an LML implementation of heap profiling, and how heap profiles can be drawn as graphs. An example shows how heap profiling can be used to dramatically improve the space behaviour of a lazy functional program.

Implementation of Evaluations Transformers

Denis Howe and Geoff Burn
Imperial College, London

Evaluation Transformers use strictness information to execute lazy functional programs more efficiently or in parallel. For example, when reducing

the expression "length l", we know that we must evaluate the structure of l. We can therefore compile code to do this rather than build a closure to represent the unevaluated list.

We introduce four evaluators for lists: E0 does no evaluation, E1 evaluates the first constructor (NIL or CONS), E2 and E3 recurse down the tail of the list and E3 also evaluates each element of the list. An evaluation transformer says how much evaluation is safe for an argument given the amount of evaluation which is safe for an application. "Safe" in this context means that evaluation will terminate if and only if lazy evaluation of the program would have.

A static program analysis annotates each application of a known function with an evaluation transformer based on the strictness properties of the function and arguments. We compile versions of each function which evaluate the result to E1, E2 or E3. If the function being applied is not known until run-time (because it is a parameter of some other function) then we may still be able to initiate evaluation of some arguments once the function is entered. Also, more evaluation of an argument may become possible at run-time than was allowed at compile-time. We associate a "current degree of evaluation" with each node in the graph so we can force more evaluation at run-time when it is safe to do so.

We have started to implement these ideas in GHC, the Haskell compiler being produced by Glasgow University. This has involved some rethinking of the original implementation proposals which were expressed in terms of the Chalmers G-Machine. GHC is based on the STG Machine where CASE expressions perform evaluation and LET expressions build closures. Function arguments are constrained to be simple variables, anything else is bound to a new variable in a LET. The idea then is to turn these LETs into CASEs where this is allowed by the evaluation transformers.

Automatic Parallelization of Lazy Functional Programs

Guido Hogen, Andrea Kindler and Rita Loogen
RWTH Aachen

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 worthwhile. 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 described by Rita Loogen at the PARLE 89 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.

Splitting infinite sets of unique names by hidden state changes

Lennart Augustsson, Mikael Rittri and Dan Synek
Chalmers University of Technology

Counter passing is the traditional method of generating unique names in a purely functional language, but it is awkward to use and imposes unnecessary data dependencies which can destroy opportunities for lazy or parallel evaluation. A method of Hancock does not have this drawback, but instead wastes large portions of the name space, so that the generated names cannot be trusted to fit inside a machine word. We implement Hancock's operations in the lazy language Haskell without wasting names, using hidden state changes à la Burton's *decisions*. Measurements show that this can be faster than counter passing even when no laziness or parallelism is gained.

Code Generation for a Lazy Functional Language

E.P. Wentworth
Rhodes University, South Africa

A native-code implementation of a lazy functional language on a SPARC architecture is discussed. The compiler produces intermediate stack-based code for a derivative of the SECD machine, the FLFM. Native code is generated by an algorithm which partially executes the FLFM code.

Some lazy list-based programs are found to exhibit particularly poor register window locality: this is partially alleviated by a hybrid policy which sometimes uses register windows and sometimes saves the context explicitly.

Good performance is achieved for those functions which do not require the full generality demanded by lazy functional languages. In particular, simple numeric functions with strict arguments have execution rates that approach the performance of the vendor's C compiler.

Straight-line code generation and evaluation by partial evaluation

Fritz Henglein
DIKU

The main problems in exploiting the fine-grain parallelism in straight-line code (basic blocks) are the small size of basic blocks in typical sequential code and the sequential data dependencies within basic blocks.

We observe that partial evaluation addresses both of these issues: partial evaluation of oblivious or partially oblivious programs (often to be found in numerical applications) results in large basic blocks (as previously observed),

and, using the algebraic properties of the operators in the resulting straight-line code, partial evaluation can be used to generate very efficient parallel code with a high degree of processor utilization.

This is exemplified by the well-known problem of parallel evaluation of expressions (and, more generally, straight-line code) over a semi-ring. In this context, partial evaluation is evaluation over the induced polynomial semi-ring that can be executed in parallel by tree contraction.

Abstract Interpretation of Parallel Functional Programs

John Hughes
University of Glasgow

Although functional programs cannot update shared data, graph reducers do. Consequently parallel graph reducers must synchronise on each access to the graph, and this synchronisation can cost 20% of the execution time. My goal is an analysis to detect possible *simultaneous uses* of values: graph nodes which cannot be simultaneously used by two parallel processes can be accessed without synchronisation. I take the task decomposition as given, and so analyse a language with explicit parallel constructs.

The approach is to define a non-standard semantics for the explicitly parallel language, and derive an analysis by abstracting it. A novel aspect is that the non-standard semantics is constructed systematically: it is a call-by-name semantics in a monad which pairs each value with a *process* representing its computation. These processes lie in an unusual and somewhat counter-intuitive process algebra: processes can continue computing after 'termination' (delivering a result), and consequently sequential composition in some cases reduces to parallel composition! Pleasantly, it is sufficient to define an abstraction of *processes* to fix an abstraction of the entire semantics.

I define analyses to detect simultaneous *uses* and simultaneous *demands*: the former is simpler, the latter permits more optimisations since only the

first use of a node updates it. Practical results are very disappointing — almost no opportunities to remove synchronisation are discovered. This is essentially because the original semantics is call-by-*name* while real implementations use call-by-*need*. When the source program is transformed to model call-by-need much better results are obtained.

The monad of functional processes offered much insight into the behaviour of parallel programs, and was definitely useful. On the other hand, call-by-name proved a very poor approximation to call-by-need. Unfortunately call-by-need is hard to abstract well: a way of doing so is urgently needed.

Computing Fixed Points in Static Program Analysis

Hanne Riis Nielson
Aarhus University, Denmark

In the context of abstract interpretation we study the number of times a functional needs to be unfolded in order to compute the least fixed point. In the monotone framework where all functions are known to be monotone we get an exponential upper bound. If we restrict the functions to be strict and additive we get a quadratic upper bound. These bounds are tight in a certain sense.

By specializing the form of the functionals we show how the classical notions of fastness and k-boundedness carry over to the framework of abstract interpretation. This gives us an alternative and more precise way of bounding the number of unfoldings needed to compute the least fixed points. For iterative forms (often obtained when analysing tail recursive programs) and primitive recursive forms (often obtained for programs with a single recursive call) we have simple methods for bounding the number of unfoldings.

Finally, we suggest an algorithm for computing fixed points based on iterative squaring. In the monotone framework it has time complexity O(log k * n) when the functional is k-bounded and the domain of the functions has

n elements. When the functions are strict and additive the time complexity can be reduced to O(log k * $(\log n)^3$).

Dependence Analysis in Array Computations

Paul H J Kelly
Imperial College, UK.

Dependence analysis of loop programs with arrays is a successful body of techniques with well-known applications in compiling imperative languages for target architectures with vector or pipelined arithmetic, cache memories and multiple processors. This talk introduced and reviewed the approach and compared it with the work of the functional programming and data flow communities. Important points of contact include

- the compilation of data parallel functional programs to make efficient use of intermediate storage.
- generating closure-free code for recursive functional array definitions.
- algebraic transformation of functional programs to enhance parallelism and locality.

I concluded by discussing work in progress on applying dependence analysis to translate loop programs into functional versions in which dependence is accurately retained, with the objective of exploiting algebraic transformation techniques.

The Structure of a Back-End to Compile Functional Languages

Manuel M.T. Chakravarty
TU Berlin

We present the structure of a back-end feasible for the compilation of lazy functional (logic) languages. It is based on the JUMP-machine and supports a wide range of optimizations. Emphasis is laid on independence from the target machine and on the use of techniques developed in the "traditional" compiler construction.

The back-end consists of three independent translation steps. The first and the second produce two different intermediate languages, namely JCode (JUMP-machine code) and TreeCode. Assembly code for the target machine is generated in the final step. The intermediate language JCode is an abstract, block-structured Code supplying an unlimited number of named variables. So, memory mapping is not done at this stage, but the functional features (e.g. pattern matching, suspensions, term construction) are explicit. In contrast, TreeCode is a linear code including labels and goto instructions. Access paths are explicit, i.e. memory mapping is done. Nevertheless independence from the target machine is preserved.

Together with optimizations like copy propagation and boxing analysis the memory mapping is performed on the level of JCode. Thereby, variables are seperated in different memory classes. Including the class of the pseudo registers that contains the candidates for later assignment to hardware registers, saving the unnecessary work of a stack simulation.

In the last translation step information depending on the target machine is introduced, e.g. number and kinds of registers. Also, code selection is performed and some final optimizations like constant folding are done.

The implementation of such a back-end led to an efficient implementation of the functional logic language Guarded TermML. Thereby, the use of the back-end generator BEG saved a lot of effort in developing the last translation phase.

Small expressions should be evaluated

Martin Raber
Universität Saarbrücken

Delayed evaluation has been recognized as one source of inefficiency when implementing a lazy functional language. One attempt for optimization has been through strictness analysis. Strict expressions can always be evaluated at once without a fear of changing the termination behaviour.

Beyond strictness analysis there is another class of subexpressions being well suited for an immediate evaluation. We call them small expressions. Small expressions only contain non-recursive functions without functional parameters. We present two abstract interpretations:

- one shows up to what extent a small expressions needs its free variables
- another one computes up to what extent a variable will be evaluated at a certain point.

Constraints + Control = Compilation or How to Compile Matching and Residuation in LIFE

Gerard Ellis and Peter Van Roy
DEC Paris Research Laboratory, France

The language LIFE is based on the underlying idea of normalizing a conjunction of primitive constraints. There are two basic operations: unification (enforcing equality of objects by adding constraints to the global state) and matching (implication, i.e. checking whether constraints are entailed or disentailed). Variables are logical variables. The data structures are ψ-terms, which are to Herbrand terms (Prolog terms) what dynamic records are to static arrays. A ψ-term can be represented as a conjunction of primitive constraints.

Functions calls in LIFE have a data-driven component. A function *fires* if the actual parameters imply the formal parameters. If *fails* if the actual parameters imply the negation of the formal parameters. In the third case the function suspends, or *residuates* (that is, a residual equation is created). The

suspended call will be reactivated if any of the residuated variables are further instantiated. Therefore it is not necessary to know a function's arguments to call it, and a function behaves like a passive constraint or demon.

This talk presents an efficient algorithm for executing matching and residuation for functions that consist of a single rule. The primitive constraints that make up the function's formal parameters form a dependency tree (which may be a rational tree). The algorithm traverses the tree and maintains a set of residuated constraints along a wavefront in the tree. Each of these constraints is the root of a subtree containing all the constraints that depend on it. Therefore a subtree contains precisely the work that must be suspended when its root residuates.

The compiled code consists of a depth-first arrangement of the constraints in the dependency tree, with a mechanism for efficiently suspending and reactivating the execution of arbitrary subtrees. This mechanism was discovered by Micha Meier, who used it to implement unification. The constraints are macro-expanded into native code, with an instruction granularity similar to WAM instructions. Our algorithm achieves linear code size and time (in the size of the formal parameters), is able to residuate on multiple constraints without blocking, does no redundant work, and does fast failure. Further work will extend the algorithm to implement functions consisting of multiple rules and to take analysis information into account.

An OR-Parallel Implementation of K-LEAF Based on Graph Rewriting

Werner Damm, Feixiong Liu and Thomas Peikenkamp
Oldenburg Universität

K-LEAF is a first-order functional + logic language based on Horn Clause logic with equality. Most of current K-LEAF implementations (sequential or parallel) rely on an extended version of Warren Abstract Machine (K-WAM). Here we present a graph rewriting model enhanced with sharing

for the OR-Parallel execution of K-LEAF programs. Our graph rewriting model provides an approach to dealing with sharing between OR-Parallel processes. With this approach, recomputation of shared computation paths can be avoided. We also address some important concepts in functional + logic models such as lazy evaluation and functional style parallelism. Lazy evaluation can be more naturally supported in our graph rewriting model than in the K-WAM because of an embedded suspension mechanism. The functional style parallelism can be exploited by combining parallel unification and strictness annotations.

A Scheme for Constraint Functional Logic Programming

Francisco-Javier Lopez-Fraguas
Universidad Complutense de Madrid, Spain

We present a general scheme CFLP(X) for first order constraint functional logic programming which plays, with respect to lazy functional logic programming with constructor discipline (as realized, e.g.,in K-LEAF or BABEL), a similar role to the well known of "Constraint Logic Programming", CLP(X), with respect to logic programming.

In CFLP(X), over a base structure equipped with a set of predefined functions and predicates, we define new ones by means of "constrained conditional rewrite rules". We formulate a declarative semantics in a general setting, where base structures are Scott domains and functions and predicates (both primitive and user defined ones) are continuous. We are able to prove that every consistent program has a least model, which is also the least fixpoint of an operator associated to the program.

We propose a sound operational semantics, constrained narrowing, which is a generalization of narrowing where unification is replaced by the more general notion of constraint satisfaction. We give then an abstract characterization of lazy computations, and we prove that the resulted computation

mechanism, lazy constrained narrowing, is complete for semantically non ambiguous programs. The proof of this is rather simple, and is based on a suitable notion of approximation to expressions and goals, for which well-founded orderings are provided.

Implementing a Lazy Functional Logic Language with Disequality Constraints

Herbert Kuchen, RWTH Aachen
Francisco Lopez-Fraguas, UCM Madrid
Juan Jose Moreno, UCM Madrid
Mario Rodriguez-Artalejo, UCM Madrid

We investigate the implementation of a lazy functional logic language (in particular the language BABEL) which uses disequality constraints for solving equations and building answers. We specify a new operational semantics which combines lazy narrowing with disequality constraints and we define an abstract machine tailored to the execution of BABEL programs according to this semantics. The machine is designed as a quite natural extension of a lazy graph narrowing machine. Disjunctions of disequalities are handled using the backtracking mechanism. The presented mechanisms are rather independent of the LBAM and can be inserted into other narrowing machines as well. If there are no disequalities in the program, the machine behaves exactly like the LBAM, and no additional overhead is needed. A prototype implementation is in progress.

FACILE: A Functional Language for Parallel Programming

Bent Thomsen
ECRC, Munich

In this talk we present the Facile language currently under development at ECRC. Facile is an experimental programming language resulting from a concrete attempt to integrate the typed call-by-value λ-calculus with a process language similar to CCS. Call-by-value λ-calculus and CCS have merged symmetrically to obtain a language that supports both functional and process abstractions: functions may be defined and used to specify internal computations of concurrent processes; dynamic process creation and synchronised communication over typed channels may occur during any expression evaluation. Functions, processes and communication channels are first class values. The language has static typing. At the theoretical level, an operational semantics has been developed in terms of labelled transition systems, and a notion of observability of programs is defined by extending the notion of bisimulation. A version of the language supporting polymorphic types has been implemented by extending the ML language with the concurrency constructs of Facile.

A New Direction in Functions as Processes

Lone Leth Thomsen
ECRC, Munich

We present the process calculus LAP (LAbel-passing communicating Processes) suitable for describing reconfigurable networks of communicating processes where the only computation is the reconfiguration of the network. The work is motivated by the many attempts to develop parallel/concurrent implementations of functional languages (e.g. by using director strings as the underlying model) and by the recent developments in process algebras for various notions of concurrency. We introduce director strings, and we translate director string expressions into LAP. The translation is done by translating the director string conversion rules into LAP. We present a reduction algorithm for rewriting LAP representations of director string expressions.

Finally we work through an example, and the correctness of the translations is demonstrated.

Process Networks and Graph Rewriting

John Glauert
University of East Anglia

Building on the work of Milner, Leth, and Honda and Tokoro, on modelling lambda-expressions as process networks, we develop a number of new translations with novel features:

- In addition to the pure lambda-calculus, we are able to translate simple data values and their operators.
- The model of communication used is essentially asynchronous and will allow a straightforward implementation.
- The process notation used may be mapped to a rewriting system in the practical graph rewriting language Dactl.

Two translations are discussed. One combines both lazy and by-value evaluation using annotated applications in the style of Burton. This translation exhibits useful parallelism.

A second translation is suitable for translating all the features of the Facile language of Giacalone, Mishra, and Prasad, which provides a symmetric integration of concurrent and functional programming. Both functional and process features of Facile are treated.

TPL : The Typed λ-Calculus with First-Class Processes

Flemming Nielson
Aarhus University, Denmark

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 take place 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.

PROGRAMMING BY MULTISET TRANSFORMATION

Daniel Le Metayer
Imperial College, London

We present a formalism, called Gamma, in which programs are described in terms of multiset transformations. A distinguishing property of Gamma is the possibility of expressing algorithms in a very abstract way, without artificial sequentiality. The expressive power of the formalism is illu- strated through a series of examples. Then we consider the problem of transforming functional programs to increase their level of parallelism. We present a technique allowing us to derive the canonical form of a program w.r.t.

the associativity/commutativity properties of certain operators. The derived canonical form is a very parallel version of the original function and it can be expressed as a combination of Gamma programs. As an aside this transformation can also be used to prove the equality of functions w.r.t. associativity/commutativity of certain operators.

Confluence Properties of Gamma Programms

Chris Hankin
Imperial College, London

Gamma is a programming notation based on multiset transformation. The attraction of the formalism is that programs exhibit a high degree of parallelism (and non-determinism) since the multiset data structure does not impose any sequentiality on access. However, the usual style of program derivation in Gamma is based on the sequential composition of multiset transformers which are instances of a small set of program schemes (Tropes). In this talk, we will study how sequential composition can be safely replaced by parallel combination of Tropes. The correctness of this transformation depends on the fact that the Tropes instances are deterministic (confluent). We establish sufficient conditions for a Gamma program to be confluent and discuss the issue of modularity of confluence.

Graph-Grammar based Monitoring of Parallel Graph Reduction Machines

Heiko Dorr
Freie Universität Berlin

Graph-grammar based monitoring extends standard event trace techniques. These techniques usually lead to significant system performance degradation even when a small amount of information is extracted during the run of the measured system. Thus, event sampling is a commonly used alternative although it gives mainly quantitative system measures.

In graph-grammar based event trace monitoring, a complete operational graph-grammar model of the system, here a parallel graph reduction machine, minimizes the performance degradation. To achieve this minimizsation, a correnpondence between a) graphs and system states and b) rewrite rules and events is set up. Since the system state is represented by a graph, each state transition, i.e. a reduction step, can be signalled by very short event messages. At least for interpretative graph reduction machines, this method seems a practical way to get necessary information for analyzing and debugging purposes.

A Distributed Garbage Collector

David R. Lester
Manchester University

A new Garbage Collector was presented which addresses the somewhat tricky problem of collecting cyclic structures within a, basically, reference counting context.

The algorithm draws heavily on previous work by John Hughes, Ian Watson, Paul Watson, and David Bevan.

The algorithm is (theoretically) ideally suited to its intended application area: implementing Functional Language on Distributed Architectures. A decision on its practicality awaits its implementation.

Dagstuhl-Seminar 9213 **Participants** (update: 25.3.92)

Lennart **Augustsson**
Chalmers University of Technology
Department of Computer Sciences
S-41296 Goteborg
Sweden
augustss@cs.chalmers.se
tel.: +46 31 721042 (from April '92: 7721042)

Manuel **Chakravarty**
TU Berlin
Fachbereich Informatik
FR 5-6
Franklinstr. 28/29
W-1000 Berlin 10
Germany
chak@cs.tu-berlin.de
tel.: +49-30-314 25213

Werner **Damm**
Universitat Oldenburg
FB 10 - Informatik
Ammerlander Herrstr. 114
W-2900 Oldenburg
Germany
Werner.Damm@arbi.informatik.uni-oldenburg.de
tel.: +49-441-798 4502

Heiko **Dörr**
Freie Universitat Berlin
Fachbereich Mathematik
Institut fur Informatik
Nestorstrae 8-9
W-1000 Berlin 31
Germany
doerr@inf.fu-berlin.de
tel.: +49-30-896 91 106

Hugh **Glaser**
University of Southampton
Department of Electronics and Computer Science
Southampton SO9 5NH
Great Britain
hg@ecs.soton.ac.uk
tel.: +44-703 54 36 70

John **Glauert**
University of East Anglia
School of Information Systems
Norwich NR4 7TJ
Great Britain
jrwg@sys.uea.ac.uk
tel.: +44-603-59 26 71

Eric **Goubault**
Ecole Normale Superieure
LIENS
45 Rue d'Ulm
F-75230 Paris Cedex 05
France
goubault@dmi.ens.fr

Kevin **Hammond**
Glasgow University
Department of Computing Science
17 Lilybank Gardens
Glasgow G12 8QQ
Great Britain
kh@dcs.glasgow.ac.uk

Chris **Hankin**
Imperial College of Science
Department of Computing
180 Queen's Gate
London SW7 2BZ
Great Britain
clh@doc.ic.ac.uk
tel.: +44-71-589 5111

Fritz **Henglein**
University of Copenhagen
Department of Computer Science
DIKU
Universitetsparken 1
DK-2100 Copenhagen
Denmark
henglein@diku.dk
tel.: +45-3139 3311 ext. 530

Guido **Hogen**
RWTH Aachen
Fachbereich Informatik
Lehrstuhl f/"ur Informatik II
Ahornstr. 55
W-5100 Aachen
Germany
ghogen@zeus.informatik.rwth-aachen.de
tel.: +49-241-80 21241

Denis **Howe**
Imperial College of Science
Department of Computer Science
180 Queen's Gate
London SW7 2BZ
Great Britain
dbh@doc.imperial.ac.uk
tel.: +44-71-589 5111 (5064)

John **Hughes**
University of Glasgow
Department of Computing Science
Glasgow G12 8QQ
Great Britain
rjmh@dcs.glasgow.ac.uk
tel.: +44-41-330 4454

Sebastian **Hunt**
Imperial College of Science
Department of Computer Science
180 Queen's Gate
London SW7 2BZ
Great Britain
lsh@doc.ic.ac.uk

Paul **Kelly**
Imperial College of Science
Department of Computer Science
180 Queen's Gate
London SW7 2BZ
Great Britain
phjk@doc.imperial.ac.uk
tel.: +44-71-589 5111 ext 5028

Herbert **Kuchen**
RWTH Aachen
Fachbereich Informatik
Ahornstr. 55
W-5100 Aachen
Germany
herbert@zeus.informatik.rwth-aachen.de
tel.: +49-241-802 1211

David **Lester**
The University of Manchester
Dept. of Computer Science
Manchester M13 9PL
Great Britain
dlester@cs.man.ac.uk
tel.: +44-61-275 5726

Lone **Leth Thomsen**
ECRC Munchen
Arabellastr. 17
W-8000 Munchen 81
Germany
lone@ecrc.de
tel.: +49-89-92 69 91 34

Feixiong **Liu**
Universitat Oldenburg
FB 10 - Informatik
Ammerlander Herrstr. 114
W-2900 Oldenburg
Germany

Francisco **Lopez-Fraguas**
Universidad Complutense de Madrid
Fac. de Matematicas UCM
28040 Madrid
Spain
tel.: +34-1-39 44 512

Daniel **le Metayer**
Universite de Rennes
IRISA
Campus de Beaulieu
Avenue du General Leclerc
F-35042 Rennes Cedex
France
lemetayer@irisa.fr

Flemming **Nielson**
Aarhus University
Dept. of Computer Science
Ny Munkegade 116
DK-8000 Aarhus C
Denmark
fnielson@daimi.aau.dk
tel.: +45-86-12 71 88

Hanne Riis **Nielson**
Aarhus University
Dept. of Computer Science
Ny Munkegade 116
DK-8000 Aarhus C
Denmark
hrn@daimi.aau.dk
tel.: +45-86-12 71 88

Eric **Nocker**
Katholieke Universiteit Nijmegen
Department of Informatica
Toernooiveld
NL-6525 ED Nijmegen
The Netherlands
eric@cs.kun.nl
tel.: +31-80-65 2509

Thomas **Peikenkamp**
Universitat Oldenburg
FB 10 - Informatik
Ammerlander Herrstr. 114
W-2900 Oldenburg
Germany
thomas.peikenkamp@arbi.informatik.uni-oldenburg.de
tel.: +49-441-798 4520

Rinus **Plasmeijer**
Katholieke Universiteit Nijmegen
Department of Informatica
Toernooiveld
NL-6525 ED Nijmegen
The Netherlands
rinus@cs.kun.nl
tel.: +31-80-65 26 43

Martin **Raber**
Universitat des Saarlandes
Fachbereich 14 - Informatik
Im Stadtwald 15
W-6600 Saarbrucken 11
Germany
raber@sol.cs.uni-sb.de
tel.: +49-681-302 2964

David **Sands**
Imperial College of Science
Department of Computer Science
180 Queen's Gate
London SW7 2BZ
Great Britain
ds@doc.imperial.ac.uk
tel.: +44-71-589 5111 Ext 4993

Helmut **Seidl**
Universitat des Saarlandes
Fachbereich 14 - Informatik
Im Stadtwald 15
W-6600 Saarbrucken 11
Germany
seidl@cs.uni-sb.de
tel.: +49-681-302 2454

Bent **Thomsen**
ECRC Munchen
Arabellastr. 17
W-8000 Munchen 81
Germany
bt@ecrc.de
tel.: +49-89-92 69 91 82

Peter **Van Roy**
Digital Equipment Corporation
Paris Research Laboratory
85 Avenue Victor Hugo
F-92500 Rueil-Malmaison Cedex
France
vanroy@prl.dec.com
tel.: +33-1-47 14 28 65

David **Wakeling**
University of York
Department of Computer Science
York YO1 5DD
Great Britain
dw@minster.york.ac.uk
tel.: (+4) 0904 432777

Peter **Wentworth**
Rhodes University
Department of Computer Science
Grahamstown 6140
South Africa
cspw@alpha.ru.ac.za
tel.: +27 41 22023

Reinhard **Wilhelm**
Universitat des Saarlandes
Fachbereich 14 - Informatik
Im Stadtwald 15
W-6600 Saarbrucken 11
Germany
wilhelm@cs.uni-sb.de
tel.: +49-681-302 3434

Zuletzt erschienene und geplante Titel:

H. Alt , B. Chazelle, E. Welzl (editors):
Computational Geometry, Dagstuhl-Seminar-Report; 22, 07.10.-11.10.91 (9141)

F.J. Brandenburg , J. Berstel, D. Wotschke (editors):
Trends and Applications in Formal Language Theory, Dagstuhl-Seminar-Report; 23, 14.10.-18.10.91 (9142)

H. Comon , H. Ganzinger, C. Kirchner, H. Kirchner, J.-L. Lassez , G. Smolka (editors):
Theorem Proving and Logic Programming with Constraints, Dagstuhl-Seminar-Report; 24, 21.10.-25.10.91 (9143)

H. Noltemeier, T. Ottmann, D. Wood (editors):
Data Structures, Dagstuhl-Seminar-Report; 25, 4.11.-8.11.91 (9145)

A. Dress, M. Karpinski, M. Singer(editors):
Efficient Interpolation Algorithms, Dagstuhl-Seminar-Report; 26, 2.-6.12.91 (9149)

B. Buchberger, J. Davenport, F. Schwarz (editors):
Algorithms of Computeralgebra, Dagstuhl-Seminar-Report; 27, 16.-20.12.91 (9151)

K. Compton, J.E. Pin , W. Thomas (editors):
Automata Theory: Infinite Computations, Dagstuhl-Seminar-Report; 28, 6.-10.1.92 (9202)

H. Langmaack, E. Neuhold, M. Paul (editors):
Software Construction - Foundation and Application, Dagstuhl-Seminar-Report; 29, 13..-17.1.92 (9203)

K. Ambos-Spies, S. Homer, U. Schöning (editors):
Structure and Complexity Theory, Dagstuhl-Seminar-Report; 30, 3.-7.02.92 (9206)

B. Booß, W. Coy, J.-M. Pflüger (editors):
Limits of Modelling with Programmed Machines, Dagstuhl-Seminar-Report; 31, 10.-14.2.92 (9207)

K. Compton, J.E. Pin , W. Thomas (editors):
Automata Theory: Infinite Computations, Dagstuhl-Seminar-Report; 28, 6.-10.1.92 (9202)

H. Langmaack, E. Neuhold, M. Paul (editors):
Software Construction - Foundation and Application, Dagstuhl-Seminar-Report; 29, 13.-17.1.92 (9203)

K. Ambos-Spies, S. Homer, U. Schöning (editors):
Structure and Complexity Theory, Dagstuhl-Seminar-Report; 30, 3.-7.2.92 (9206)

B. Booß, W. Coy, J.-M. Pflüger (editors):
Limits of Information-technological Models, Dagstuhl-Seminar-Report; 31, 10.-14.2.92 (9207)

N. Habermann, W.F. Tichy (editors):
Future Directions in Software Engineering, Dagstuhl-Seminar-Report; 32; 17.2.-21.2.92 (9208)

R. Cole, E.W. Mayr, F. Meyer auf der Heide (editors):
Parallel and Distributed Algorithms; Dagstuhl-Seminar-Report; 33; 2.3.-6.3.92 (9210)

P. Klint, T. Reps (Madison, Wisconsin), G. Snelting (editors):
Programming Environments; Dagstuhl-Seminar-Report; 34; 9.3.-13.3.92 (9211)

H.-D. Ehrich, J.A. Goguen, A. Sernadas (editors):
Foundations of Information Systems Specification and Design; Dagstuhl-Seminar-Report; 35; 16.3.-19.3.9 (9212)

W. Damm, Ch. Hankin, J. Hughes (editors):
Functional Languages:
Compiler Technology and Parallelism; Dagstuhl-Seminar-Report; 36; 23.3.-27.3.92 (9213)

Th. Beth, W. Diffie, G.J. Simmons (editors):
System Security; Dagstuhl-Seminar-Report; 37; 30.3.-3.4.92 (9214)

C.A. Ellis, M. Jarke (editors):
Distributed Cooperation in Integrated Information Systems; Dagstuhl-Seminar-Report; 38; 5.4.-9.4.92 (9215)