Distribution in a Demand Driven Style

Ian Holyer, Neil Davies and Eleni Spiliopoulou
Department of Computer Science
University of Bristol
{ian,neil,spilio}@cs.bris.ac.uk

Abstract

In this paper we present a model of distribution as a natural consequence of deterministic concurrency; a purely declarative form of concurrency that preserves referential transparency. This extends the demand driven model used as the operational basis of functional languages, into one with multiple independent demands without the need to allow non-determinism to appear at the user level.

The abstract model of this distributed system uses a uniform access memory model in order to provide a global memory view. This, combined with explicit thread based concurrency and explicit representation of demand enables the trustable distribution of reactive systems across different systems. Since referential transparency is preserved, any pattern of distribution has no effect on the semantics of the functional program. Computation, including both code and data, can be moved in response to demand. This suggests itself as a natural approach to facilitate mobile computation both as a conceptual and operational model.

1 Introduction

Distributed applications become more and more pervasive since they play a major role in the current trends of computing and represent a significant proportion of computer software. Such applications include the following classes as identified by Bal [2, 23].

- parallel and high-performance applications;
- fault-tolerant applications and real-time systems;
- distributed operating systems;
- inherently distributed applications such as email and World Wide Web applications.

In this paper we focus on supporting mechanisms for inherently distributed applications. Distributed applications involve many independent components, possibly located at different places, which cooperate in order to provide a single integrated computer facility and enable users to share resources via a network. The expansion of the Internet requires that distributed applications evolve further, allowing the "mobility" of applications, while minimising the impact of hardware issues. Such mobile software applications aim at providing online services, interactive distance learning, home networking etc.

We present a new conceptual model for building distributed applications which derives as a natural extension to a purely functional form of concurrency, which is called deterministic concurrency; its most important characteristic is that it retains referential transparency and does not allow non-determinism to appear at the user level. The main virtue of this system is that applications that use it are trustable in the presence of distribution. This trustability derives from its ability to be repeatable and resilient, a property that emerges from the preservation of semantics through deterministic concurrency. Furthermore, this distributed system allows for mobile computation since it exhibits performance flexibility, i.e. it maximises the scope for evaluation independent of location and can naturally express distribution, migration and mobility.

2 Purely Functional Distributed Systems

2.1 Concurrency and Distribution

There are two levels in computing where the question of concurrency arises. The first, often described as parallelism, indicates the use of multiple agents or processes in the internal representation of a program, in a way which does not affect its semantics or its externally visible behaviour, except in terms of efficiency. The expressive power of languages though, is not increased beyond sequential execution; although many activities may be carried out at the same time, they are coordinated in such a way that the results are the same as if the program was executed sequentially. Functional programming languages have been considered suitable for parallel execution due to their operational freedom in evaluation order and a lot of research has been carried out in this field, but we don't pursue these ideas further since their are out of the scope of this paper.

At the second level, concurrency denotes the ability of a program to do several things at once in an asynchronised way. A simple test whether a language has concurrency or not is weather it can solve the following problem. Take two existing interactive processes, for example two processes running in two separate subwindows of the program's main window, and combine them into a single program. The program should run both processes at the same time, with no interference between them. If a user presses a key or mouse button in one subwindow and starts off a long computation, it should still be possible for him to interact with the process in the other subwindow while the computation is proceeding. For the rest of the paper the term concurrency is used to indicate explicit, concurrent `I/O` performing processes.

Distributed applications require concurrency in their underlying implementation to increase their expressiveness, since it allows multiple users to access services without waiting for one another. To provide concurrency, programming languages have had concurrency primitives introduced into them which create, control and allow processes to cooperate. However, they introduce non-determinism as a consequence of timing dependencies during process creation and communication. Multiple processes may share the same state, with non-deterministic interleaving of accesses. Alternatively, they may communicate by passing messages and non-deterministically merge them to a common destination. As a consequence, concurrent systems exhibit non-deterministic behaviour. This can manifest itself as undesired behaviour under certain, usually rare, sequences of events. It also adds an extra burden of complexity onto the programmer, making it very difficult to assert such systems as reliable and resilient.

Since concurrency is a necessary requirement of distribution, all existing distributed systems suffer from complexity and unpredictability, due to timing elements introduced by non-determinism. All this conspires to make them even more difficult to debug, adapt and maintain.

2.2 Deterministic Concurrency

The effects of non-determinism on functional languages are particularly damaging and thus, it has been considered problematic to introduce concurrency into them. Non-determinism destroys their clean properties, notably referential transparency, and therefore equational reasoning can no longer be applied [8]. Despite these restrictions, Haskell [1] has been similarly adapted to support concurrency by having such concurrency primitives added [10]. To restore semantic properties, a compromise has been adopted, where non-determinism is still accepted as a "necessary evil" but the internal referential transparency in the language is retained, as explained by Finne and Peyton Jones [9]. Effectively, non-determinism is pushed out of the functional program into the `I/O` actions and `I/O` state so that the programs are internally referentially transparent. The externally visible behaviour of such programs though, becomes non-deterministic and thus less predictable and repeatable. This is a successful approach for a single self-contained program within a conventional procedural operating system. However, this approach is not sufficient in the case of a distributed program built from communicating components, where a program must be deterministic as a whole, not just each of its parts. This requires to bring communication inside the language and making it inherently referentially transparent, which is a key issue to allow for arbitrary migration of computation without any change in the meaning of a program.

The need for a truly declarative approach to concurrency prompted further research in the functional programming community. Holyer and Carter [6, 12] discussed that a deterministic approach to concurrency can be achieved by switching from a conventional, *data driven* to a *demand driven* approach. Under the data driven approach, a program must be able to respond to events coming from multiple input streams, according to which event arrives first, which requires a non-deterministic merge. Under the demand driven approach, multiple input streams to a program correspond to multiple independent threads (demands). The arrival of an event in one stream wakes up the relevant thread and produces some output for that thread. This can be implemented without any non-deterministic merge. The demand driven approach [6, 12] offers a purely declarative model of concurrency which is called *deterministic concurrency*. Deterministic concurrency is a form of concurrency which is based on the non-strict functional programming evaluation model. In this model expressions are evaluated on an "as-needed" basis and computation evolves in order to satisfy a single requirement, or demand, that of providing some output to the external world. This is captured in the `IO` monad [26], which guarantees sequencing of the external requirements. Deterministic concurrency extends the single demand execution model into a model with multiple independent demands and has been added in such a way that the declarative nature of the semantics is preserved.

Although such a demand driven approach to concurrency results in a restricted form of concurrency, compared say to the π-calculus [22], it enables the behaviour of reactive systems to be expressed in purely functional terms. This provides the ability to serve multiple external requirements in a way that they don't conflict so that interaction of inputs and outputs happens in the right order. Since referential transparency is preserved, output values and their order becomes fully defined by the code and trace of inputs. Under the deterministic concurrency approach, a processor can run one or more `I/O` performing operations without including any user-visible non-determinism. This is guaranteed by the independence of demands, e.g. demands are associated with distinct devices or windows. Moreover, the issues raised by such a declarative form of concurrency can be of interest in their own right, beyond functional programming, and can be adopted in any language or `I/O` model.

To investigate the expressiveness of this form of concurrency, a prototype implementation of a single-user multi-threading environment [6] has been built and the following simplifications were undertaken in order to focus on the basic ideas:

- A fixed mapping of processes onto processors;
- Statically defined communications interfaces;
- Separate local memories.

This prototype implementation has been proved expressive enough so that a complete single-user multi-tasking working environment can be built upon it. As a consequence, it demonstrated that deterministic concurrency is expressive enough to build a complete single-user multi-threaded environment. It also proved that most of the non-determinism apparent in traditional concurrent systems is unnecessary and can be eliminated by minor redesigns. Furthermore, based on the principles of deterministic concurrency, Serrarens [27] built BriX, a deterministic concurrent windowing system.

Deterministic concurrency has been added in the Brisk compiler [1], as part of the Brisk system [13], acting as a substrate for building concurrent applications the behaviours of which can be expressed in purely functional terms.

2.3 Deterministic Distribution

To address the issue of distribution, concurrency is a necessary prerequisite. Since we did not want to compromise referential transparency, we investigated distribution as a natural extension of deterministic concurrency. Within the semantic framework that has been developed for deterministic concurrency, distribution is almost an orthogonal issue. Systems can be distributed in a trustable way, without any change to the "meaning" of programs – the only observable difference is the speed at which the overall system generates results.

Based on the results of the above mentioned feasibility study, we removed the initial set of restrictions listed above, allowing the pattern of processes and their interconnections to evolve, allowing code as well as data to migrate among processors, and having a global memory model to give such migration a clear meaning. This provides a conceptual framework which allows for arbitrary distribution of computation in a "demand driven" style [30], since the semantics of programs are preserved independently of how a program is distributed. In this framework, the preservation of referential transparency offers a trustable form of distribution which ensures reliability and effective reuse of software. In other words, deterministic concurrency extends the call-by-need model of lazy evaluation into one with multiple independent demands and deterministic distribution enables these independent demands to reside in separate processors. It also offers the flexibility to choose dynamically whether unevaluated code fragments or result values are communicated. This allows for new approaches in load balancing and issues of value security and integrity. In this framework all entities are mobile; computation, both data and code, can be moved dynamically on demand in an entirely arbitrary manner. This distributed model provides a number of important characteristics, most of them based on the fact that distribution is orthogonal to functionality.

- It allows us to treat any mechanisms such as distribution mapping, processor identification, or granularity of the distribution purely as annotations to the program. An annotation is an explicit indication in the representation of a program which changes its operational semantics, but not its meaning. Distributing the program becomes an operation that can

[1] Brisk stands for Bristol Haskell Compiler

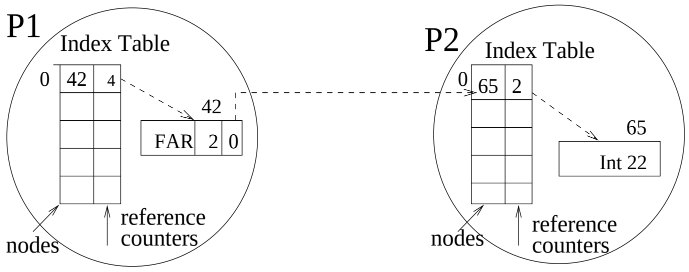

Figure 1: The Global Memory model

be performed late in the development life-cycle and the programmer is freed from the burden of explicitly interweaving all the distribution details within program code.

- It makes a clear distinction between state, communication and functionality, which further allows the separation of the semantics and design of programs from their distribution. This is in sharp contrast to traditional distributed programming models [29] (client-server, RPCs), where all the distribution mechanisms have to be programmed explicitly.

- This distributed model provides also a high level of trustability. This is guaranteed on the one hand by the deterministic concurrency and on the other by the fact that distribution is an issue orthogonal to the functionality of a program. Deterministic concurrency guarantees the correctness of a concurrent program in an one-processor setting. The fact that distribution is orthogonal to the design of the program ensures safety in any pattern of distribution.

- Distributed programs can be easily debugged. A program can be developed on one processor, where reasoning and debugging are relatively easy before being distributed, i.e. one can test a program on one machine and run it over many machines. Proof of functional correctness apply equally to the program, whether running as a single processor system or as a distributed system.

- It provides an approach to distributed problem solving which enables the implementation of reactive systems, the "behaviours" of which remain predictable and are independent of how they are distributed. Referential transparency ensures that in a distributed setting, output values happen in the right order, since they are fully defined only by the input code and trace.

3 The Operational Model

Functional programming languages are well known to have a close relationship between operational and denotational semantics. Our approach to distribution was to introduce operational activities in form of annotations that have no effect on the declarative semantics. An annotation is equivalent to an identity function, i.e. it is semantically inert and at the same time able to change the operational behaviour of a program.

The operational model for distribution uses a uniform access method in order to provide a distributed, transparent, addressing scheme and to enable global knowledge of the current state of the system. This, combined with both explicit thread based concurrency with no user exploitable non-determinism and explicit representation of external demand provide the mechanism to distribute reactive systems on several processors and offers a new model for computational mobility. As we have totally preserved referential transparency, any pattern of distribution has no effect on the denotational semantics of the system.

This operational model has a number of important characteristics, most of them based on the fact that distribution is orthogonal to functionality:

Global Memory Model The operational model implements a global distributed heap. It consists of a collection of processors which forms a network, as in figure 1. Processors communicate by exchanging messages, but this is not a message passing approach. Messages are automatically generated by the run-time system and are totally hidden from the applications programmer. Each processor contains heap allocated objects which hold part of the overall graph as specified by annotations to the program. The complete set of heaps can be thought of as forming a single global memory which contains the complete graph of the program.

Each processor includes an index table. The index tables deal with inter-processor references. They are important in our design, because if an inter-processor reference contained the absolute address of a node in the target processor, it would be difficult to move that node during a local garbage collection. Instead, such a remote reference contains an offset within the index table of the target processor, and this in turn contains the absolute local address of the node. This decouples the local heap from the global distributed heap and enables global knowledge of the current state of the system. In addition, the index tables provide a reference counting scheme by recording the number of remote references to each node, which helps to keep the distributed graph consistent and allows for local garbage collection asynchronous from the rest of the system.

A reference to a remote piece of graph is an indirection that acts as a synonym for that node. Such a non-local reference contains two pieces of information. The first one indicates which processor is the owner of the required piece of graph and the second references an entry point in the index table of the remote processor. This entry point refers to the required node as well as to a reference counter indicating the number of remote references to this object. When an indirection to a remote node is encountered during execution, the value of the remote node needs to be requested. A new entry in the local index table is added, referring to the local indirection node, which acts as a target for the reply. This index table entry is recorded in a request message and sent to the remote processor. The processor returns the requested node (or an evaluated version of it or an extended section of graph) in a reply message to the requester together with the necessary information on how to access the target of the reply through the index table. The reply message on arrival accesses through the index table entry the node that initiated the request, overwrites it with the result and decrements the reference counter of the index table entry.

Deterministic Concurrency It is achieved through lightweight threads which do not share any state; their states are independent of each other. Threads communicate only via shared constant values, which they return as results (which may be lazily evaluated lists of items produced during execution of a process). A process which attempts to evaluate a piece of graph locks it during evaluation to prevent any other process from duplicating the work of evaluation unnecessarily. When a particular piece of graph is fully evaluated, its value can be shared at will.

Explicit Representation of Demand Another feature of our distributed operational model is the explicit representation of demand to initiate remote graph reduction. In functional languages, evaluation is forced by the demand for the value of a graph node. This is also called call-by-need semantics. In a sequential program, there is a single source of demand which causes graph reduction. In a concurrent setting the graph is evaluated concurrently by allowing more than one source of demand. In a distributed setting, multiple sources of demand may reside in separate processors. Messages carry demand across processor boundaries; when a message requesting a node arrives at a processor, it causes a new thread to be created, which represents a new demand forcing a local piece of graph to be evaluated. Control of execution moves dynamically from processor to processor, driven by messages. From the point of view of an external observer of the execution all we have introduced are differences in the relative timings of values generated by the overall system, not the values themselves or their order. Tracing the execution of a thread becomes equivalent to observing the passage of demand and computation through the distributed graph.

Migration of Computation As already explained, this distributed system allows for variable granularity of distribution which is specified by explicit annotations at arbitrary points of the program graph to indicate which portion of it should be evaluated locally and which remotely. Thus, the unit of distribution can vary from individual values, through graph fragments (closures) of individual functions, to complete modules. This provides the underlying operational infrastructure to support migration as a mechanism to satisfy system constraints, e.g. performance, security. Migration happens by allowing both flow of data and flow of demand through messages. The flow of demand can be specified in two ways: whether the owner of a piece of the execution graph returns the required sub-graph for evaluation, or evaluates it and returns its value; there are two bases for making this choice.

The first is a question of load balancing; if the owner of the subgraph is more heavily loaded than the requester, it is better to return the unevaluated subgraph and allow the requester to do the evaluation. This also acts as a general mechanism for controlling the granularity of mobile code and data. Load balancing thus is orthogonal to the distribution mechanism and several techniques from parallel programming can be adapted safely.

The second is a question of communication overhead; evaluating a subgraph before returning it may result in fewer messages overall. In addition, evaluating expressions in advance before they are requested (either speculatively or because strictness information shows that they will be needed) acts as a mechanism which tolerates latency. Traditionally in functional languages, distribution has been considered as providing support for parallelism and thus, tolerating latency has been a major issue in improving efficiency. This distributed system is not aimed at providing a high-performance parallel application [25, 16, 7, 32], but a new model of distribution and computational mobility for loosely coupled systems. However, tolerating latency is also a major issue since a distributed system needs always to respond quickly. As already explained, introducing additional demand which speculatively evaluates expressions allows additional threads to be introduced, so that data can be evaluated in advance and sent in bulk on demand to the requester.

Highly Available Distribution The global memory model of distribution provides the potential support for long-lived entities, i.e. programs and data that last for a complete user session, or are part of a permanently running operating system environment. On the other hand, long-lived entities are threatened by long-term garbage. Cycles which span processors lead to degradation of such programs, but it is notoriously difficult to deal with in interactive systems where it is not practical to bring the entire system to a halt in order to collect garbage.

We outline here an hierarchical, non-blocking, distributed garbage collection scheme which deals asynchronously with garbage. This allows for maximising garbage collection efficiency under the assumption that most garbage is locally defined, i.e. garbage collection is performed within the local area more frequently. This approach also allows the isolation of non-available portions of the memory while partial garbage collection occurs.

Collecting distributed non-cyclic garbage is based on a variant of Lester's distributed garbage collection algorithm [17]. The reachable set of nodes in a distributed setting consists of any nodes referenced internally as well as any active nodes that reside remotely. The latter are recorded in the index table and their reference counter is non-zero. If such remote reference becomes garbage, i.e. its reference counter becomes zero, this information is propagated to the owner of this piece of graph and so it can be garbage collected.

For collecting cycles without to bring the system to a halt, we propose a scheme for collecting cyclic garbage or non-available sectors similar to the one proposed by Maheshwari and Liskov [20]. It consists in forcing cycles or entities which might have not been touched for a long time to migrate into a single processor, where local garbage collection techniques can be used. This approach isolates non-available sectors on a variable scale as needed simultaneously with partial garbage collection.

Sharing In keeping with the traditional functional paradigm, sharing ensures uniqueness of unevaluated objects in a distributed environment. This is achieved either by distributing references or by giving away a piece of graph and replace its local copy with a reference. Operationally, sharing is preserved by the index table through a reference counting scheme which records the number of remote references to a particular piece of graph. It also ensures that the distributed graph is consistent, i.e. equivalent to a non-distributed graph after performing the same number of steps. It also shows that a distributed program is equivalent to a program within a single processor, with the sole exception that not all of the unreachable memory (garbage) has been recovered. Evaluated objects, i.e. values are infinitely sharable and can be freely copied.

4 Mobility

Mobility is the ability of a language to change dynamically the location of execution of programs. This distribution model has two important characteristics which make it mobile: first, it allows to ship code fragments with variable granularity and second, it is referentially transparent.

Cardelli [4] argues that computation is not just code but also the context of its execution. In Obliq [3], mobility is realised by moving closures containing the context of the computation. Our distributed system has the potential to support variable granularity of mobility, thus it enables the communication of any portion of the graph, together with any associated context of its execution. This is as well realised by moving closures, since their use is common in functional language implementation. Heap objects are stored as closures containing references to code for executing them; these heap objects contain portions of the execution graph, which can either be data, values or computation. We have extended the concept of closure within a distributed context; marshaling suitable portions of graph in messages allows code and its associated context to be communicated. The implementation of this distributed system is based on the Brisk Machine [14], a variant of the `STG` machine [24]. The Brisk Machine is especially designed to be flexible and dynamic so that it can support a uniform and efficient implementation of multiple paradigms such as computational mobility, dynamic loading and linking and logic programming in the form of the Escher [19] programming language. Furthermore, the Brisk Machine provides a dynamic code model where functions are stored as heap nodes. This supports distributed execution and mobility of computation, where code is communicated dynamically between machines, currently in interpretive bytecode format to ensure portability.

The preservation of referential transparency in a distributed environment has further implications, always orthogonal to the denotational semantics.

First, it provides a low latency distribution scheme, i.e. programs can start running without complete code; the missing code can be replaced by remote references as a form of dynamic binding. This derives from the fact that purely functional languages do not require a high degree of global consistency within the memory model, a property which is kept in a distributed environment since referential transparency is preserved.

Second, it results in a location independent distributed system. Through referential transparency, a distributed computation given the same inputs will always produce the same outputs, in the same order, independently of the pattern of distribution. The only difference will be the relative timings of which these outputs appear. Computation can be moved on demand and the location of computation can change at run-time.

4.1 Global Computation

The global address space scheme which forms the operational basis of this distributed system offers an alternative approach to global computation. This is mainly due to the unique naming scheme it offers. As a natural consequence, it raises all the programming issues described by Cardelli [5]. We analyse them in the context of our model:

Typing. This mobile computation model is based on the non-strict purely functional paradigm and thus has the rich, polymorphic type system of Haskell. Type correctness is guaranteed on the one hand by static checking and on the other hand by the unique naming of objects in the presence of distribution.

Security. Cryptographic security is an important problem for distributed computation. Although sufficiently strong bulk encryption mechanisms already exist, the main challenge is to ensure the authentication of the end-points. Authentication of end-points needs to be resilient in presence of masquerade attacks. In this distributed system, uniqueness of end-points is ensured by the the unique naming of objects in the global address space scheme, e.g. ASN object identifiers. These unique end-points can be used in conjunction with a trusted directory service.

Reliability. This model has the underlying features that support reliability, e.g. the possibility of replication of values. In addition, the distributed garbage collection scheme we adopt provides support for isolating the non-available portions of memory, e.g. through temporary system unavailability.

Modularity. Modularity is needed to increase the potential for dynamic computation, i.e. to enable components to migrate at run-time. Wadler [33] argues that in functional languages, the explicit data flow together with the monadic approach provide the ultimate form of modularity, which is preserved in our distributed environment.

Resource Management The global address scheme provided in this distributed system enables the unique naming of objects and therefore provides a better resource management scheme. Linking to facilities in this distributed system happens via a global identifier and not just by name, which guarantees that the other end obtains a compatible version.

Resilience One further important programming issue in the context of global computation is resilience, which ensures trustability and repeatability in a distributed environment. In this distributed system, there is no possibility to introduce timing related dependencies when an application is distributed. This comes together with referential transparency which allows for equational reasoning. Whether the final distributed application performs its tasks at an acceptable rate is a different question, at least the question of value correctness and ordering does not exist.

5 Related Work

Erlang [34] is a concurrent functional language also used for distributed applications; although its distribution mechanism has been based on traditional concepts, its model does not need an interface description language and so it facilitates considerably the distribution process. Distributed Erlang is used in implementing large software projects within the Ericsson group.

Distributed Corba [28] interfaces traditional models into a distributed environment through the Interface Definition Language `IDL`. `IDL` is for Corba what `HTML` is for the Web and therefore it does not address any mobility issues.

Java [11] supports weak mobility, i.e. the unit of distribution is fixed at design time. This is necessary because Java is implemented on top of traditional communications mechanisms which allow for non-deterministic effects and therefore fixing the unit and location of computation at design time makes distribution semantically visible in denotational terms.

The Inferno [15] model has been designed as an operating system to be used in network environments. It provides interface to resources, scattered across the network and so distributed applications can be created. In Inferno, the interface to resources is fixed at design time, whereas the location of the code is statically assigned at run-time, i.e. it is the interface to resources that defines the granularity and the mobility.

Tarau and Dahl [31] presented a mobile system implemented on top of the BinProlog compiler, enriched with Linda primitives and a continuation passing approach. Linda primitives differ though from our approach since they provide concurrency as a means of increasing the efficiency of a language and not its expressiveness. Furthermore, the Linda model is highly non-deterministic; processes interact via a tuple space. The programmer has to ensure that such non-determinism doesn't give rise to unacceptable answers.

Obliq [3] supports strong mobility. It has both a graph metaphor and the ability to move demand. Obliq objects are local to the site of creation therefore marshaling of computation needs to happen at design stage.

Icarus [21] also supports strong mobility and has been designed as an extension to Occam [18]. Icarus preserved Occam's small granularity but removed its fixed topology with mobile channels. Although it is a dynamic model, it does not provide variable granularity of mobility, once assigned it is fixed.

6 Conclusions

In this paper, based on the belief that the properties of functional languages allow programs to be well distributed, we presented a purely declarative model for distribution of computation based on the non-strict functional paradigm. Distribution has been built on top of deterministic concurrency; a purely functional form of concurrency which extends the demand driven model of execution of functional languages into one with multiple independent demands without the need to allow non-determinism to appear at the user level. Since referential transparency is retained in this distributed model, computation can be moved arbitrarily either according to demand or other management reasons. Moreover, this scheme provides a new framework for computational mobility where software, both its code and data can migrate dynamically.

We have a prototype implementation which is currently being incorporated into the Brisk run-time system.

References

[1] Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnson, Mark Jones, Erik Meijer, Simon L. Peyton Jones, Alastair Reid, and Philip Wadler. Haskell 1.4, A Non-strict, Purely Functional Language. Technical report, Yale University, 1997.

[2] Henri Bal, Jennifer Steiner, and Andrew Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[3] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[4] Luca Cardelli. Mobile Computation. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems*, volume 1222 of *LNCS*, Linz, Austria, July 1996. Springer Verlag.

[5] Luca Cardelli. Global Computation. *Sigplan Notices*, Position Paper, 1997.

[6] David Carter. Deterministic Concurrency. Master's thesis, Department of Computer Science, University of Bristol, September 1994.

[7] Manuel M.T. Chakravarty. A Self-Scheduling, Non-Blocking, Parallel Abstract Machine for Lazy Functional Languages. Proceedings of the 6th International Workshop on the Implementation of Functional Languages, 1994.

[8] W. Clinger. Nondeterministic call by need is neither lazy nor by name. In *Conference on Lisp and Functional Programming*, 1982.

[9] S. Finne and Simon L. Peyton Jones. Programming Reactive Systems in Haskell. In *Glasgow Functional Programming Workshop*, Ayr, Scotland, 1994. Springer-Verlag.

[10] S. Finne and Simon L. Peyton Jones. Concurrent Haskell. In *Principles of Programming Languages*, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM Press.

[11] J. Gosling, J. B. Joy, and G. Steele. *The Java Language Specification.* Addisson Wesley, 1996.

[12] Ian Holyer and David Carter. Concurrency in a Purely Declarative Style. In John T. O' Donnell and Kevin Hammond, editors, *Glasgow Functional Programming Workshop*, pages 145–155, Ayr, Scotland, 1993. Springer-Verlag.

[13] Ian Holyer, Neil Davies, and Chris Dornan. The Brisk Project: Concurrent and Distributed Functional Systems. In *Glasgow Functional Programming Workshop*, Ullapool, July 1995. Springer-Verlag.

[14] Ian Holyer and Eleni Spiliopoulou. The Brisk Machine: A Simplified STG Machine. In Chris Clack, Tony Davie, and Kevin Hammond, editors, *9th International Workshop on the Implementation of Functional Languages*, volume 1467 of *LNCS*, pages 20–38, Saint-Andrews, Scotland, September 1997. Springer Verlag.

[15] Lucent Technologies Inc. *Inferno Programmers Guide.* Murray Hill, NJ, 1997.

[16] H. Kingdon, D.R. Lester, and G.L. Burn. The HDG-machine: a highly distributed graph-reducer for a transputer network. *The Computer Journal*, 34(4):290–301, Aug 1991.

[17] D.R. Lester. An efficient distributed garbage collection algorithm. In E Odijk, M. Rem, and Syte J.C., editors, *Parallel Architectures and Languages Europe*, volume 365 of *LNCS*, pages 207–223, Eindhoven, The Netherlands, June 1989. Springer Verlag.

[18] Inmos Limited, editor. *Occam programming manual.* Prentice-Hall, 1984.

[19] J. W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[20] Umesh Maheshwari and Barbara Liskov. Collecting Cyclic Distributed Garbage by Controlled Migration. *Distributed Computing*, 10(2):79–86, 1997.

[21] David May and Henk L Muller. Icarus language definition. Technical Report CSTR-97-007, Department of Computer Science, University of Bristol, January 1997.

[22] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I,II. *Information and Computation*, 100(1):1–77, 1992.

[23] Paddy Nixon, Vinny Cahill, and Fethi Rabhi. Building Parallel and Distributed Systems. *The Computer Journal*, 40(8):463–464, 1997.

[24] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.

[25] Simon L. Peyton Jones, Chris Clack, and Jon Salkild. High-performance parallel graph reduction. In E Odijk, M. Rem, and Syte J.C., editors, *Parallel Architectures and Languages Europe*, volume 365 of *LNCS*, pages 193–206, Eindhoven, The Netherlands, June 1989. Springer Verlag.

[26] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *Principles of Programming Languages*, Charleston, January 1993. ACM Press.

[27] Pascal Serrarens. BriX - A Deterministic Concurrent Functional X Windows System. Master's thesis, Department of Computer Science, University of Bristol, June 1995.

[28] Jon Siegel. *CORBA Fundamentals and Programming.* John Wiley and Sons, 1996.

[29] Eleni Spiliopoulou. An Analogy between Client-Server, RPC's and a Functional Deterministic Distributed System. In *Spring Research Conference*, Bristol, April 1997. Faculty of Engineering.

[30] Eleni Spiliopoulou, Ian Holyer, and Neil Davies. Distributed Programming, a purely functional approach. In *International Concference of Functional Programming*, Amsterdam, The Netherlands, June 1997. ACM Press. Poster Abstract.

[31] Paul Tarau and Veronica Dahl. Mobile Threads through First Order Continuations. In *Proceedings of APPAI-GULP-PRODE'98*, Coruna, Spain, jul 1998.

[32] Philip Trinder, Kevin Hammond, J. Mattson, A. Partridge, and Simon L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Programming Languages Design and Implementation*, Philadelphia, USA, 1996. ACM Press.

[33] Philip Wadler. Monads for functional programming. *Lecture notes for Marktoberdorf Summer School on Program Design Calculi, Springer-Verlag*, Aug 1992.

[34] Claes Wikstrom. Distributed Programming in Erlang. In *PASCO'94, First International Symposium on Parallel Symbolic Computation*, Linz, Austria, September 1994.