Motivation for Glasgow distributed Haskell, a non-strict Functional Language

P.W. Trinder

Heriot-Watt University, Edinburgh, EH14 4AS, Scotland
E-mail: trinder@cee.hw.ac.uk

Non-strict functional languages offer potential benefits for constructing distributed systems: namely a highly-dynamic model of distribution, a relatively high degree of distribution transparency, and the potential to abstract over distribution-control primitives. We describe our motivation for implementing such a language, a variant of Haskell, and evaluating it. The implementation is a fusion of existing Glasgow Haskell Compiler technologies. The evaluation will be based on experiences implementing a distributed interactive simulation, and comparing it with a Java version.

1 Introduction

Many applications, particularly those with multiple users are most naturally structured as a collection of processes distributed over a number of machines. Examples include multi-user games and simulations, distance learning tools, and software development environments. The advent of the world wide web has stimulated the development of distributed applications, and the technology to support them [5,6,9,12,28].

Large-scale distributed applications may have components written in multiple languages, supplied by several vendors and executed on a heterogeneous collection of platforms. They also have elaborate mechanisms for managing the failure of components of the system. Large-scale distribution is supported by standard interfaces like CORBA [31] or Microsoft DCOM [22], and entails considerable machinery. In contrast, small-scale distribution entails components written in a single language, is typically constructed by a single vendor, and is often restricted to an homogeneous network of machines, with a simple model of failures. Small-scale distributed applications are typically constructed in a distributed programming language, e.g. Java with Remote Method Invocation (RMI) [6]. Distributed languages allow the system to be developed in a single, homogeneous, framework, and make the distribution more transparent to the programmer. In consequence a distributed language is less cumbersome than a standard interface like CORBA.

2 Distributed Languages

2.1 Taxonomy

There are many distributed languages, and it is not possible to cover all of them in detail. Instead we outline a framework in which to locate our work. Two key concepts in distributed languages are processes and threads.

- A *process* has a private name (or address) space. Processes synchronise and communicate explicitly, using a variety of mechanisms, e.g. channels in Occam [29], or RMI in Java [6]. Data is typically explicitly and statically allocated to a process.

- A *thread* (or lightweight thread) shares a name space with other threads and communicates using shared data. For example Java threads may share a class of objects and communicate using synchronised methods.

Threads and processes may be dynamically or statically allocated to processing elements (PEs). Distributed languages can be classified in a lattice depending on their support for threads and processes, as depicted in Figure 1. The classes of language are as follows.

- *Implicitly-parallel* languages have no explicit processes or threads but execute on multiple processors, e.g. HPF [13], NESL [1]. These languages are not suitable for distribution, but have parallelism implicit in control or data structures.

- *Shared name space* languages support threads, but not processes, e.g. OZ [12], Oblique [5], GpH [33] and Linda [8].

- *Distributed name space* languages support processes but not threads, e.g. Facile [9], PICT [28], and languages with a communications library like C and MPI [23].

- *Shared and distributed name space* languages support both threads and processes. Threads may reside in a processes name space, as in Java [6], or may not, as in Occam [29].

Conventional distributed languages like Java or C-Split [6] provide high-level support for communication and synchronisation, e.g. remote procedure calls (RPCs) and synchronised methods. They typically have explicit, static task and data partitioning, although some recent languages now support dynamic task and data placement, for example Oz [12]. Every value communicated

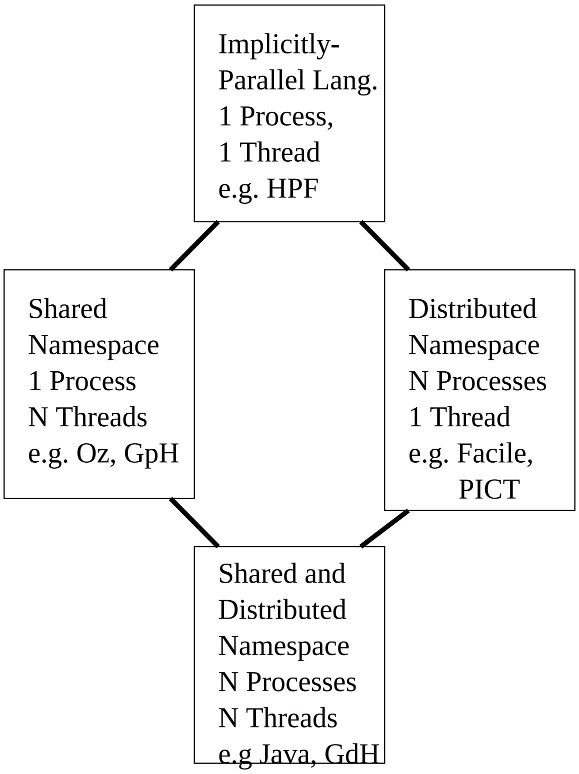

Figure 1: Classes of Distributed Programming Language

between processes is sent explicitly, and must be fully evaluated prior to transmission. Hence values transmitted must be first order, i.e. functions and computations cannot be transmitted. Programs are non-deterministic and the programmer is responsible for avoiding problems of deadlock and starvation.

2.2 Distributed Functional Languages

Distributed functional languages typically support a more dynamic approach. For example allowing dynamic data and task placement as in Kali Scheme [5], and the dynamic generation of processes and channels, as in Eden [2]. They also support the transmission of higher-order values. Some languages even have deterministic semantics, like Brisk [11].

In a non-strict distributed language data transfer between processes can be lazy and dynamic. The cost of laziness is an additional message from the recipient requesting the data, there are several specific benefits:

- Lazy transfer is useful if part of a large (or even infinite) data structure are to be exchanged. Logically the entire data structure is exchanged, but the receiving process will only demand as much of the data structure as is needed. In practice, to offset the high communications latencies of distributed systems transfer is at least partly eager. For example a packet-sized chunk of a data structure may be sent when the first element is demanded.
- Lazy transfer avoids the problem of a fast producer flooding a slow consumer's memory.
- Dynamic transfer is useful if the amount of data to be sent is hard to determine *a priori*, or varies between program executions.

Each of these issues can of course be explicitly managed in a strict context, but explicit management is cumbersome and error-prone.

We intend to investigate these costs and benefits in the context of realistic distributed programs. The approach of evaluating novel language technology by implementing real programs has proved valuable for both Concurrent Haskell and GpH [7,21]. We will also make our language available to research collaborators. Collaborators with a particular interest are the Eden Project, at the University of Marburg, for comparison with their Eden distributed language [2]; and the UniForM project, at the University of Bremen for use in their program development environment [17]

3 GdH: Glasgow distributed Haskell

Haskell [26] is the de facto standard lazy functional language, and the project is based on the Glasgow Haskell Compiler (GHC), arguably the best lazy functional language implementation in existence [27]. As Figure 2 depicts, GHC supports two modest extensions to sequential Haskell: Concurrent Haskell and Glasgow Parallel Haskell (GpH). Indeed the construction of a distributed functional language in a short period is only possible because it is a fusion of these two mature implementations.

GpH supports multiple threads on multiple PEs, but has only a single process. Threads are introduced and synchronised using parallel and sequential composition. Evaluation strategies abstract over the compositions to allow polymorphic and higher-order description of parallel behaviours [33], e.g. parList creates a thread to evaluate every element of a list.

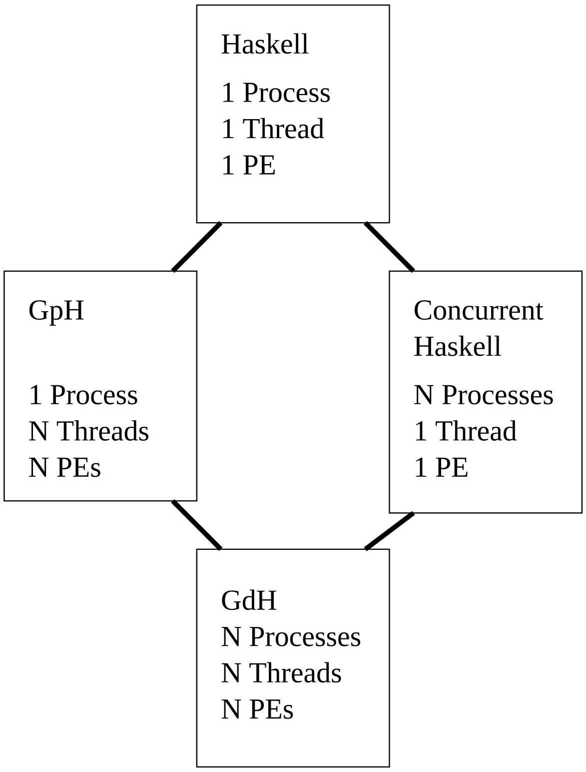

Figure 2: Haskell Variants

Concurrent Haskell supports multiple processes on a single PE. Processes are created by a monadic `forkIO` primitive, and communicate using `Mvars`, which are akin to M-structures in Id [3].

Our proposed language, Glasgow distributed Haskell (GdH) is a straightforward fusion of the GpH and Concurrent Haskell. Threads and local processes can be created as before. A new monadic primitive, `rforkIO` allows the creation of process on a remote PE, possibly identified by an abstract processor identifier. The operational semantics of an `Mvar` is extended to support distributed communication. We hope to be able to abstract over the primitives to allow high-level description of distributed behaviours in an analogous fashion to evaluation strategies. The distributed programming model is akin to that of Java in supporting both threads and processes. With the exception of `Mvars` and `rforkIO` processes, the distribution of all values is transparent.

The potential benefits of GdH are as follows.

- GdH combines threads and processes in a highly-dynamic model of distribution. In particular thread and process creation, thread and data placement and data transfer are all dynamic. Moreover data transfer is lazy.

- Most aspects of the distribution in GdH are transparent and automatically managed by the runtime system. The notable exception is the remote processes created by `rforkIO`.

- GdH is based on a small number of primitives. We hope to be able to abstract over them to provide higher-order, polymorphic control of distribution, along the lines of evaluation strategies [33]. In the longer term we hope to be able to give formal semantics to the primitives, and reason formally about program as in PICT [28], and GpH [10].

- GdH will be supported by a robust, architecture-independent implementation, allowing the construction of realistic programs to evaluate the costs and benefits of the language.

4 Evaluating a Distributed Functional Language

The implications of GdH for small-scale distributed programming are best investigated by constructing a real, non-trivial application that makes essential use of distribution. A suitable demonstration application could come from a range of application areas, including programmer support environments [17] distance learning environments, games [4] or simulations [19]. A multi-user distributed interactive simulation is chosen as a demonstrator on the basis of the availability of expertise, and of a suitable example.

Kiteck's factory simulation [19] is chosen as the simulation to be distributed because it is real, of a suitable size, and has been previously studied. The simulation identified the cause of a bottleneck in a distribution facility. The advantage of distributing the simulation is that it allows several experts, e.g. production manager, inventory controller and operations researcher, to cooperate in the investigation of production-line problems without needing to be in the same place. Kiteck's factory simulation does not require elaborate simulation tools like sophisticated random number generators, and visualisation tools, as evidenced by existing implementations that do not use a simulation package (like C-SIM [30]). The existence of implementations in other languages supporting small-scale distribution e.g. Java with RMI[35] will allow direct comparison of the languages. For example we will be able to make comparative measurements of the communication costs of both implementations.

The evaluation will focus on the benefits of the highly-dynamic model of distribution and the relatively high degree of distribution transparency available in GdH. Before leaving the topic of simulation we should clarify that distributed interactive simulation is used in the military sector specifically to mean systems with both real and simulated components, with tight real-time synchronisation. For example in a distributed interactive simulation of an army manoeuvre soldiers and tanks on the ground interact with simulated aircraft. Such systems are large-scale, and are supported by an elaborate, standard interface, the High Level Architecture[14]. In stark contrast the distributed interactive simulation we propose is small-scale, and synchronises events within a few seconds rather than milliseconds.

5 Future Plans

The investigation will proceed in four main phases, as described below. The first phase of the project is to design GdH, as a modest extension of Haskell. It is anticipated that the design will be a straightforward fusion of GpH and Concurrent Haskell. The second phase is to implement modest extensions to the GHC Haskell compiler, and GUM runtime system to support GdH. During this phase we will utilise our expertise with both compiler and runtime system Once constructed we will make an alpha-release of GdH publicly available. In particular we will make it available to the Eden group at Marburg and the UniForM group at Bremen, and support them to use it. The third phase is to construct a distributed, interactive application, most probably Kiteck's factory simulation. We will make direct comparisons with implementations of Kiteck's simulation in other distributed languages. In the final phase, we will assess the costs and benefits of our highly-dynamic distributed programming model. Our evaluation will be based on our experiences implementing the simulation, the results of comparisons with the simulation in other languages, and the experiences of the Eden and UniForM groups.

6 Conclusion

To investigate the costs and benefits of non-strict functional languages in a distributed context we plan to implement and evaluate a distributed variant of Haskell. The implementation will be a fusion of existing Glasgow Haskell technologies. The evaluation will be based on experiences implementing a distributed interactive simulation, and comparing it with a Java version.

Acknowledgments

This work funded by EPSRC Grant number GR/M55633 "Functional Distributed Interactive Systems", and British-German Academic Research Collaboration Project number 1097 "Evaluating Functional Distribution".

References

1. G.E. Blelloch, "Programming Parallel Algorithms", *Comm. ACM*, **39**(3), 1996, pp. 85–97.
2. S. Breitinger, R. Loogen, Y, and Ortega-Mallen, "Towards a Declarative Language for Parallel and Concurrent Programming", in Proc IFL95 - Intl Workshop on Implemention of Functional Languages, Bastad, Sweden, , September 1995.
3. P.S. Barth, R.S. Nikhil, and Arvind, "M-Structures: extending a parallel non-strict functional language with state" in Proc. of Funtional Languages and Computer Architecture, Boston, , Springer-Verlag LNCS 523, pp. 538-568, September 1991.
4. N. Bartlett, S. Simkin, and C. Stranc, "Cutting Edge - Java Game Programming", Coriolis Group Books, 1998.
5. H. Cejtin, S. Jagganathan, and R. Kelsey, "Higher-order Distributed Objects", ACM Trans. On Programming Languages and Systems (TOPLAS), 17(5), September 1995.
6. M.C. Daconta, A. Saganich, E. Monk and M. Snyder, Java 1.2 and JavaScript for C and C++ Programmers, John Wiley & Sons, New York, 1998.
7. S. Finne, and S.L. Peyton Jones, "Composing Haggis", Proc. of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics, Maastricht, September 1995.
8. D. Gelernter, and N. Carriero, "Coordination Languages and Their Significance", *CACM*, **32**(2), February, 1992, pp. 97–107.
9. A. Giacalone, P. Mishra and S. Prasad, "Facile: a symmetric integration of concurrent and functional programming. Proc. Tapsoft89, Springer-Verlag LNCS 352, pp. 181-209, 1989.
10. J.G. Hall, C. Baker-Finch, P.W. Trinder, and D.J. King, "An Operational Semantics for Parallel Lazy Evaluation", in Proc IFL98 - 10th Intl Workshop on Implemention of Functional Languages, UCL, London, UK, to appear in Springer Verlag LNCS, September 1998.
11. I. Holyer, and E.Sipiopoulou, "Concurrent Monadic Interfacing", Proc IFL98 - 10th Intl Workshop on Implemention of Functional Languages, UCL, London, UK, to appear in Springer Verlag LNCS, September 1998.

12. S. Haridi, P. Van Roy, and G. Smolka, "An Overview of the Design of Distributed Oz", in Proc. 2nd Intl. Symposium on Parallel Symbolic Computation (PASCO 97), ACM, New York, USA, 1997.
13. High Performance Fortran Language Specification, Technical Report, Rice University, Texas, USA, 1993.
14. Department of Defence, "High Level Architecture for Simulations, Version 1.1 Interface Specification", February 1997.
15. K. Hammond, H.-W. Loidl and A. Partridge, "Improving Granularity for Parallel Functional Programs: a Graphical Winnowing System for Haskell", Proc. 1995 Conf. on High Performance Functional Computing (HPFC '95), Denver, Co., April 1995, pp. 208-221.
16. P. Hudak and B. Goldberg, "Distributed Execution of Functional Programs Using Serial Combinators", IEEE Computer, 34(10):881-891, October 1985.
17. E. W. Karlsen, S. Westmeier, "Using Concurrent Haskell to Develop User Interfaces over an Active Repository", Proc. IFL'97, Implementation of Functional Languages, St. Andrew, Scotland, Springer Verlag LNCS 1467, September 1997.
18. D.J. King, J.G. Hall, and P.W. Trinder "A Strategic Profiler for Parallel Haskell", in Proc IFL98 - 10th Intl Workshop on Implementation of Functional Languages, UCL, London, UK, to appear in Springer Verlag LNCS, September 1998.
19. P. Kiteck, "Analysis of Component Interaction in a Distribution Facility using Simulation", Proc. EUROSIM, 1992.
20. U. Klusik, Y. Ortega-Mallen, and R. Pena, "Implementing Eden, or Dreams become Reality", in Proc IFL98 - 10th Intl Workshop on Implementation of Functional Languages, UCL, London, UK, to appear in Springer Verlag LNCS, September 1998.
21. H.-W. Loidl, P.W. Trinder, C.V. Hall, K. Hammond, S.B. Junaidu, and S.L. Peyton Jones, "Engineering Parallel Symbolic Programs in GpH", submitted to Concurrency: Practice and Experience, September 1998.
22. L. Merrick, "DCOM Technical Overview", Microsoft White Paper, 1996.
23. The MPI Forum, "MPI-2: Extensions to the Message-Passing Interface", Tech. Report, University of Tennessee, Knoxville, July 1997.
24. S.L. Peyton Jones, C. Clack, and J. Salkild, "High-Performance Parallel Graph Reduction", Proc. PARLE '89 – Conf. on Parallel Architectures and Languages Europe, Springer-Verlag LNCS 365, pp. 193-206, 1989.
25. S.L. Peyton Jones, A. Gordon, and S. Finne, "Concurrent Haskell", Proc. POPL'96 - 23rd ACM Symposium on the Principle of Programming Languages, St Petersburg beach, Florida, USA, January 1996.

26. J.W. Peterson, K. Hammond (eds.), L. Augustsson, B. Boutel, F.W. Burton, J.H. Fasel, A.D. Gordon, R.J.M. Hughes, P. Hudak, T. Johnsson, M.P. Jones, E. Meijer, S.L. Peyton Jones, A. Reid and P.L. Wadler, "Report on the Programming Language Haskell: a Non-Strict Purely Functional Language, Version 1.4", April 1997.
27. S.L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P.L. Wadler, "The Glasgow Haskell Compiler: a technical overview", in Proc. Of the UK Joint Framework for Information Technology, Technical Conference, Keele, pp. 249-257, 1993.
28. B.C. Pierce, D.N. Turner, "Pict: A Programming Language Based on the Pi-Calculus", Indiana University Tech. Report, 1997.
29. D.Pountain and D.May A Tutorial Introduction to Occam Programming McGraw-Hill, New York, 1987.
30. H.D. Schwelman, "C-SIM: A C-Based Process-Oriented Simulation Language", Proceedings of the 1986 Winter Simulation Conference, pp. 387-396, December 1986.
31. J. Siegel et al., "CORBA Fundamentals and Programming", John Wiley and Sons, Ney York, USA, 1997.
32. P.W. Trinder, E. Baryy Jr., M.K. Davis, K. Hammond, S.B. Junaidu, U. Klusik, H.-W. Loidl and S.L. Peyton Jones, "GpH: An Architecture Independent Functional Language", submitted to IEEE Trans. On Software Engineering, July 1998.
33. P.W. Trinder, K. Hammond, H.-W. Loidl and S.L. Peyton Jones, "Algorithm + Strategy = Parallelism", Journal of Functional Programming, 8(1), January 1998.
34. P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge and S.L. Peyton Jones, "GUM: a Portable Parallel Implentatiion of Haskell", Proc. 1996 ACM. Conf. on Programming Language Design and Implementation (PLDI '96), Philadelphia, May 1996, pp. 79-90.
35. P.A. Wilcox, "Object-Oriented Factory Simulation in Java", M.Sc Project Report, University of Edinburgh, Dept of Computer Science, 1997.