Extended Synopsis

Because the speed increases possible with single processors have reached their limit, concurrent and parallel programming are becoming indispensable. To keep up with hardware progress, software would need to approximately double from now on the amount of parallelism every 18 months. Hardware is also expected to become more *heterogeneous*, with several processors of different capabilities working together (e.g., processor/GPU combinations). Finally, computation will become more *distributed* with applications executed "in the cloud" by a demand-driven number of servers. These new demands on software have been identified as the "Popular Parallel Programming" (PPP) grand challenge by the computer architecture community [64].

The challenge is hard to meet because concurrent and parallel programming are fundamentally difficult. The conventional view is that one is left with two choices. The first possibility is to have programs manage their degree of concurrency explicitly through threads or processes. This results in a state-space explosion which tends to overwhelm the capability to understand software's behavior, even if standard locks are replaced by some higher-level synchronization mechanism such as transactions or messages. Furthermore, the non-determinism inherent in many concurrent applications means that common testing techniques become ineffective, and that software failures become very hard to reproduce and track down.

The alternative is to let the compiler find sources of parallelism automatically, transforming a sequential program into a parallel one while maintaining its semantics. This seems to be more appealing from a correctness point of view, provided a sufficient degree of parallelism can be uncovered. Functional programming languages have an advantage here because they tend to have a larger degree of implicit parallelism due to the absence of side-effects. But even in the best of circumstances, it has turned out to be very hard to uncover the necessary degree of parallelism in programs. Successes are usually limited to highly regular data sets and are often extremely brittle to scale. Using explicitly parallel code [86] can help to some degree but requires relatively high effort and expertise.

The current consensus is that non-deterministic concurrency is too dangerous on a large scale and that parallelization is not effective enough to scale most applications to large numbers of cores. However this holds only if we restrict our focus to general purpose programming languages. Individual domains have often an "embarrassingly large" degree of parallelism. Examples of such domains are machine learning, probabilistic reasoning, large-scale data acquisition, mesh-based solvers, graphics, climate simulation, event propagation in reactive programming, and many others more. Given enough time and effort, it's possible to develop highly parallel applications in these spaces. Examples are Google's massively parallel infrastructure for search or Facebook's handling of large-scale social graphs. Impressive as these systems are, the question remains how to package the expertise they embody so that more programs can be parallelized with reasonable effort. The natural way to achieve this packaging of expertise is through a *domain-specific language* (DSL) which provides high-level ways to express domain algorithms together with a set of transformations into primitives that parallelize well.

But there remain two problems with the parallel DSL approach: first, bringing all these new languages to a sufficient degree of maturity is an enormous effort. This investment would have to include not just language specifications and construction of their optimizing compilers and libraries, but also all the other aspects of modern tooling including IDEs, debuggers, profilers, build tools as well as all aspects of documentation and training. It is hard to see how such an investment can be made time and time again for each specialized domain. Second, DSLs do not exist in a vacuum but have to connect to other parts of a system. For instance, a climate simulation program could have a visualization component that is based on a graphics DSL. How to connect the two?

In this project we will develop solutions to these problems that make pervasive parallelism exploitable with reasonable effort. The principal new idea is to combine polymorphic embeddings with domain-specific optimizations in a staged compilation process. In a first step, we will express DSLs as high-level libraries in a common host language. This solves the tooling and interoperability problems. We will use Scala as a host language because it already has a track record of successfully embedding DSLs, ranging from actors [56, 50, 51, 55] and query languages [76, 9], to combinator parsing [41, 108], natural language processing[68], and machine learning[3]. We will apply the technique of *polymorphic embeddings* to represent a DSL in a host language in an abstract way. This technique provides a very elegant way to reify domain language elements and to use the reified representations in a staged compilation process.

Polymorphic embeddings are explained in detail in the full proposal. The essential idea is to represent parts of a DSL with parameterized types such as `Rep[T]`. Here, `T` is a normal host language type whereas `Rep` is an abstract type constructor called a *higher-kinded type*, which stands for the range of possible implementations of a domain type. For instance, here is a power function in a hypothetical DSL that makes use of `Rep` types.

```
def power(x: Rep[Double], n: Int) = if(n==0) 1.0 else x * power(x, n-1)
```

Instead of concrete implementations of `Rep` types one provides to a DSL the signatures of all operations that can be performed on instances of these types. In the example above, these operations would have to include the multiplication operation and an implicit injection of the constant 1.0 into the `Rep[Double]` domain. The full proposal shows how to define these operations while keeping considerable syntactic flexibility.

The implementations of all operations on `Rep` types are hidden just like the type `Rep` itself. This has important benefits. First, a DSL designer can choose the right primitives for a DSL and can prevent elements of the host language from leaking into the DSL. Second, because the `Rep` type constructor is abstract, one has complete freedom in choosing its implementation. One is not constrained by the one-size-fits-all standard approach of publishing abstract syntax trees [18].

Polymorphic embeddings for DSLs were first explored by Hofer et al. [60], building on work by Moors and myself on higher-kinded generics [84] and on work by Carette, Kiselyov and Shan on tagless staged interpreters [19]. The new direction taken by this proposal is to combine polymorphic embeddings with optimizing rewritings in a staged compilation process. Staging lets a program generate another program as its result; traditionally [48] it is expressed in terms of special pairs of syntactic brackets that separate code running in different stages of a program's execution. Polymorphic language embeddings lead to a more flexible and lightweight approach to staging, both in terms of syntactic and runtime overheads. Unlike for traditional staging, no special syntax is needed. Instead, we use the higher-kinded `Rep` types to determine the binding times of variables, which in turn determine when a piece of program is executed: Everything tagged with a `Rep` type is assumed to be dynamic, whereas every expression tagged with a normal type is static, and thus acts as a constant for the final program generation. Such staging time constants play an important role for optimizations. For instance, given a standard DAG representation of `Rep` together with common subexpression elimination and knowledge about the associativity of `*`, one can automatically rewrite a call such as `power(x, 5)` to the following sequences of instructions:

```
val x1 = x * x; val x2 = x1 * x1; x2 * x
```

It should be noted that analogous optimizations apply even if the exponent is not a compile-time constant, and that all this can be done without any help from the host language compiler, in a purely library based solution. The full proposal shows that, with just a little more work, a naive FFT implementation directly obtained from the Cooley-Tukey recurrences can be turned into an efficient butterfly network.

Traditional staging typically produces program code that must again be passed through the host language's compiler before it can be executed. Staged pieces of code are assembled by purely syntactic replacement, which means that inserting a complex expression into two "holes" duplicates that computation. By contrast, our approach is semantic, allowing to define both generic and specific rules on how to combine staged code fragments. A polymorphic embedding can be combined with arbitrary internal data structures to represent a domain language, precisely because the domain program itself is parametric in these structures. It thus allows a large range of translation and optimization strategies.

A typical usage scenario is that a piece of DSL code together with the implementation of some of the DSL API is optimized and translated at run time into a set of class files and the resulting code is executed using dynamic class loading. Alternatively, the same DSL is translated into a sequence of GPU instructions in CUDA [27], which are then loaded and executed into the processor's GPU. Yet another output option is to translate to C code to be run on a cluster with MPI [86]. The result of executing that code can then be used further in the running program, including as an argument to further staging steps.

In a sense it means we run a high-level JIT compiler, where library-defined transformations are applied to domain-specific code, and that code is translated to heterogeneous hardware, just before the result of that code is needed by the program. The usual JIT techniques such as caching and hot-spot detection all apply.

Another interesting aspect of my proposal is that the technique used for DSL embeddings can in essence be re-used for exploiting heterogeneous hardware. For instance, one can define a type `GPU[T]` for expressions that are executed in a co-processor, whereas `Local[T]` would mark expressions executed on the main processor. Either type would be produced from a `Rep[T]` with a domain-specific transformation. The abstract `GPU` type family would support exactly those operations that can be executed on a GPU, and no others. The concrete implementation of `GPU` (which again can be hidden from user programs) would then take care of GPU code generation and organize the data movement between processors.

Yet another application of polymorphic embeddings is in distributed and cloud computing. Here, one concern is to minimize the number of round trips between communicating computers. By reserving the type `Remote[T]` for expressions executing on the remote communication partner one can use staging to implement a form of batching [63], which combines as many remote computation inputs and outputs as possible into single messages.

In summary, polymorphic embeddings are a breakthrough idea for exploiting parallelism in domain languages and for modeling heterogeneous hardware. To solve the PPP challenge, we need to realize this idea in a comprehensive research effort that takes a number of practical DSLs and maps them to a range of hardware platforms. The research needs to integrate work on embeddings with work on parallel data structures and algorithms, as well as with static analysis, optimization, and code generation for heterogeneous execution platforms. This is what I propose to do.

We will collaborate in the project closely with the groups of Prof. Olukotun, Hanrahan, and Aiken in Stanford University's Pervasive Parallelism Lab. They will embed a number of their parallel DSLs into Scala using our polymorphic embedding technique. They will also provide their scheduling technology and code generators for CUDA and C/MPI to the project.

The research, summarized in Figure 1, consists of the following parts.

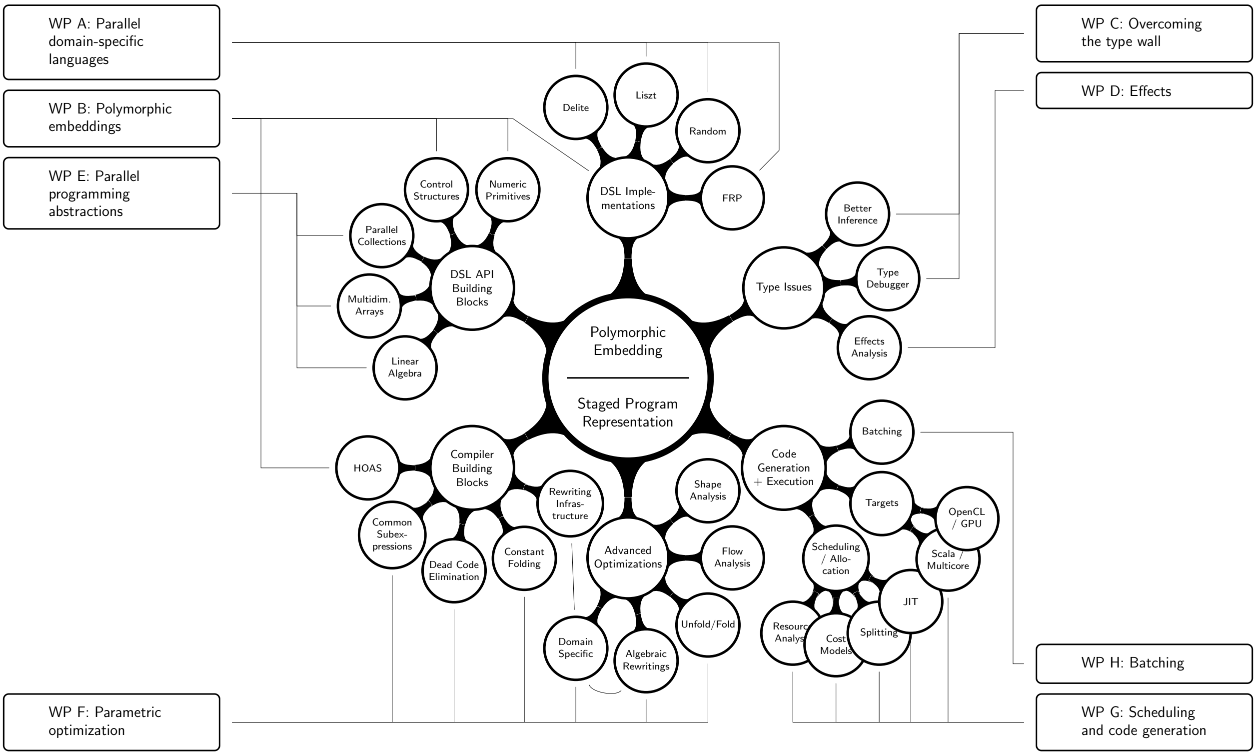

Figure 1: Work items and their relationships

WP A: Parallel domain-specific languages. We start with a number of existing DSLs. Some of these will come from the PPL work (Liszt, Random[T]), and we will also integrate two DSLs ourselves. One of these will cover reactive programming, for the other we will be looking for interested partners from other EPFL schools.

WP B: Polymorphic embeddings. We will work with Stanford to construct polymorphic embeddings of these languages into Scala, focussing on language support to generalize and refine these embeddings. This will validate the results of the work packages on types, effects, and parallel programming abstractions. We will also study the combination of different DSLs in one host program.

WP C: Overcoming the type wall. Polymorphic language embeddings put a heavy load on type formalisms, but not every application programmer is well versed in type theory. In this work package we will investigate novel techniques and tools that help ordinary programmers make effective use of advanced type systems. We will work on new schemes for type inference, on high-level type refactorings, and on new presentation mechanisms that help clarify type derivations and type errors.

WP D: Effects. A common strategy of a DSL embedding is to fall back on the host language for some parts of the domain language. For instance a query language such as Microsoft's LINQ allows syntactically arbitrary boolean expressions in `Where` clauses. However, one might want to disallow expressions whose evaluation can have side effects, can throw exceptions or might not terminate. All these aspects can be summarized as *effects*, and can be described in a common framework. The goal of this work package is to come up with a general, user-extensible effect system that's at the same time accurate and lightweight. This is not a trivial undertaking; "taming effects" has been identified as "the next big programming challenge" by Simon Peyton Jones [66].

WP E: Parallel programming abstractions. Many different DSLs share common data structures for parallel programming. In this work package we will provide a range of data structures and optimized implementations for commonly used types. Starting with parallel collections, we will also investigate types that make some use of staging. A prime example known from high-performance computing are generalized arrays over complex regions that describe valid index sets. The mapping from complex index to linear relative address can be highly optimized by representing the generalized array as a bundle of static region descriptor and dynamic data.

WP F: Parametric optimization. Once we have polymorphic encodings, we need to translate them for concurrent executions. A crucial step for this is optimization. We plan to build a modular optimization framework. Starting with common subexpression elimination, dead code elimination, and constant folding (which are all more powerful than usual in the presence of staged execution), we plan to investigate more aggressive optimization schemes such as software pipelining and supercompilation. We will also develop a parametric flow-analysis for DSLs, which produces from a DSL description with primitive flow dependencies the full dependency graph of a DSL program.

WP G: Scheduling and code generation. From the optimized representation of Scala/DSL code we generate code for multiprocessors. This code generation path will in the end produce JVM bytecodes. Crucial issues here are a good scheduling of tasks to processors and a good allocation of resources to threads. Two other code generation paths will target clusters and GPUs. We plan to describe cluster and GPU code using polymorphic embeddings analogous to our DSL encodings. The decision whether code should map to the main (multi-)processor or to a co-processor could be made initially explicitly in the DSL. But we will also investigate a splitting transform which partitions a program into code to be run on the main processor and code to be run on the co-processors. The splitting transform would need resource analyses,*e.g.*to make sure that co-processor code can be executed without recourse to dynamic memory allocation (which is generally unavailable on GPUs).

WP H: Batching. We will work on deploying DSL code on remote processors as well as on combining multiple data items into larger packages so that the number of round-trips between communicating parties is kept small. Batching is important in the co-processor context because the time spent setting up a co-processor program with new data and extracting computed results can be high relative to the cost of computation. An essentially analogous problem happens in distributed computing, where communication delays also often dominate computation costs.

The following are some important success criteria for the project: (1) Can parallel DSLs be expressed in a high-level, user-oriented way? (2) Is the host-language embedding lightweight and natural? (3) Can different DSLs be integrated easily with each other? (4) Can significant parts of the optimizer framework be re-used between DSLs? (5) Can a DSL be targeted efficiently to different hardware platforms? And, most importantly, (6) Can sufficient parallelism be extracted and efficiently exploited to make all of this worthwhile?

Overall, this is a very ambitious project where many parts have to work well both individually and together, so that the objective can be reached. But what I find exciting is that for the first time I see a chance that we can solve the popular parallel programming challenge. The project won't lead to a silver bullet that will magically parallelize arbitrary programs. But if the success criteria are met it will provide a foundation to get there with continuous work. As new application domains come up, one can put effort into designing good parallel embedded languages for these domains and use our framework to map them to parallel execution environments. Likewise, as future parallel hardware evolves one can extend our framework to accommodate new architectures. In summary the project is bound to have a major impact on the field of computing. If its objectives are met it will revolutionize the ways we approach parallel programming and domain-specific languages.

References

[1] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.

[2] Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, 2007. ACM.

[3] Ethem Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, October 2004.

[4] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In Radhia Cousot and David A. Schmidt, editors, *SAS'96, Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66. Springer, September 1996.

[5] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time Java memory management. *Real-Time Syst.*, 37(1):1–44, 2007.

[6] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1993.

[7] Ioana Banicescu and Vijay Velusamy. Load balancing highly irregular computations with the adaptive factoring. In *IPDPS*. IEEE Computer Society, 2002.

[8] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 86–95, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Alan Beaulieu. *Learning SQL, 2nd edition*. O'Reilly Media, Inc., 2009.

[10] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[11] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The development of the emerald programming language. In Barbara G. Ryder and Brent Hailpern, editors, *HOPL*, pages 1–51. ACM, 2007.

[12] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[13] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[14] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000. `http://www.javaseries.com/rtj.pdf`.

[15] Ian Buck, Tim Foley, Daniel Reiter Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[16] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 71–80, New York, NY, USA, 2007. ACM.

[17] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[18] Charlie Calvert and Dinesh Kulkarni. *Essential LINQ*. Addison-Wesley Professional, 2009.

[19] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2007.

[20] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[21] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3):291–312, 2007.

[22] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, 1990.

[23] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. Deconstructing reo. In *Proc. FOCLASA*, 2008.

[24] Ian Clarke. Swarm-dpl, a transparently scalable distributed programming language. `http://code.google.com/p/swarm-dpl`.

[25] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *In European Symposium on Programming*, pages 294–308, 2006.

[26] Microsoft Corporation. DirectCompute PDC HOL. `http://code.msdn.microsoft.com/directcomputehol`.

[27] NVIDIA Corporation. NVIDIA CUDA programming guide version 3.0. `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide_3.0.pdf`.

[28] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[29] Martina Danková and Martin Stepnicka. Fuzzy transform as an additive normal form. *Fuzzy Sets and Systems*, 157(8):1024–1035, 2006.

[30] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[31] Advanced Micro Devices. ATI Stream SDK v2.01 documentation. `http://developer.amd.com/GPU/ATISTREAMSDK/pages/Documentation.aspx`.

[32] Dominic Duggan and Frederick Bent. Explaining type inference. In *Science of Computer Programming*, pages 37–83, 1996.

[33] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(03):455–481, 2003.

[34] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

[35] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[36] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *Proc. ECOOP*, pages 273–298, 2007. Springer LNCS 4609.

[37] J. McGraw et. al. SISAL: Streams and iterators in a single assignment language, language reference manual. Technical Report M-146, Lawrence Livermore National Laboratory, March 1985.

[38] Calin Cascaval et.al. Software transactional memory: why is it only a research toy? *ACM Queue*, December 2008.

[39] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, January 1996.

[40] Matteo Frigo. A fast fourier transform compiler. In *PLDI*, pages 169–180, 1999.

[41] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In Paul Hudak and David Scott Warren, editors, *PADL*, volume 4902 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2008.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented design*. Addison-Wesley Reading, MA, 1995.

[43] Pablo Giambiagi and Gerardo Schneider. Memory consumption analysis of Java smart cards. In *Proceedings of CLEI'05*, Cali, Colombia, October 2005.

[44] George Giorgidze and Henrik Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In *Functional and (Constraint) Logic Programming, 19th International Workshop, WFLP 2010, Madrid, Spain, January 17, 2010*, 2010. To Appear.

[45] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[46] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. *SIGPLAN Not.*, 37(5):282–293, 2002.

[47] Khronos OpenCL Working Group. The OpenCL specification version 1.0. `http://www.khronos.org/registry/cl`.

[48] Miguel Guerrero, Edward Pizzi, Robert Rosenbaum, Kedar Swadi, and Walid Taha. Implementing dsls in metaocaml. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 41–42, New York, NY, USA, 2004. ACM.

[49] Christian Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *Sci. Comput. Programming*, pages 284–301. Springer-Verlag, 2003.

[50] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In David E. Lightfoot and Clemens A. Szyperski, editors, *JMLC*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006.

[51] Philipp Haller and Martin Odersky. Actors that unify threads and events. *Coordination Models and Languages*, pages 171–190, 2007.

[52] Chris Hankin and Igor Siveroni, editors. *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*. Springer, 2005.

[53] Tim Harris, Simon Marlow, Simon Peyton Jones, , and Maurice Herlihy. Composable memory transactions. In *Proc. PPoPP*, 2005.

[54] Martin Henz, Gert Smolka, and Jörg Würtz. Oz - a programming language for multi-agent systems. In *IJCAI*, pages 404–409, 1993.

[55] Carl Hewitt. Actorscript(tm): Industrial strength integration of local and nonlocal concurrency for client-cloud computing, 2009.

[56] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[57] James E. Hicks, Derek Chiou, Boon Seong Ang, and Arvind. Performance studies of id on the monsoon dataflow system. *J. Parallel Distrib. Comput.*, 18(3):273–300, 1993.

[58] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[59] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 12(10), October 74.

[60] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In Yannis Smaragdakis and Jeremy G. Siek, editors, *GPCE*, pages 137–148. ACM, 2008.

[61] P. Hudak. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142, 1998.

[62] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, 1992.

[63] Ali Ibrahim, Yang Jiao, Eli Tilevich, and William R. Cook. Remote batch invocation for compositional object services. In Sophia Drossopoulou, editor, *ECOOP*, volume 5653 of *Lecture Notes in Computer Science*, pages 595–617. Springer, 2009.

[64] M.J. Irwin and J.P. Shen, editors. *Revitalizing Computer Architecture Research*. Computing Research Association, dec 2005.

[65] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS*, volume 2 of *LIPIcs*, pages 383–414. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[66] Simon Peyton Jones. Taming effects with functional programming. Presentation given at QCon, London, mar 2009.

[67] Guy L. Steele Jr. Parallel programming and parallel abstractions in fortress. In *IEEE PACT*, page 157. IEEE Computer Society, 2005.

[68] Daniel Saul Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall, Upper Saddle River, NJ, 2 edition, 2008.

[69] Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In Giorgio C. Buttazzo, editor, *EMSOFT*, pages 249–258. ACM, 2004.

[70] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[71] Monica S. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *PLDI*, pages 318–328, 1988.

[72] Doug Lea. A Java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[73] Sean Lee, Vinod Grover, Manuel M. T. Chakravarty, and Gabriele Keller. GPU kernels as data-parallel array computations in Haskell, 2009.

[74] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL: Proceedings of the 2 nd conference on Domain-specific languages: Austin, Texas, United States*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 1999.

[75] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[76] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009.

[77] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, 1988.

[78] B. McCloskey and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *Proc. POPL*, pages 346–358, 2006.

[79] Matthew Might and Olin Shivers. Environment analysis via delta cfa. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 127–140. ACM, 2006.

[80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[81] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[82] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[83] Eugenio Moggi and F. Palumbo. Monadic encapsulation of effects: a revised approach. *Electr. Notes Theor. Comput. Sci.*, 26, 1999.

[84] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In Gail E. Harris, editor, *OOPSLA*, pages 423–438. ACM, 2008.

[85] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, K.U. Leuven, 2008. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.abs.html`.

[86] MPI Forum. MPI: A message-passing interface standard. Version 2.2, September 4th 2009. available at: `http://www.mpi-forum.org`.

[87] Martin Odersky. Functional nets. In *Proc. European Symposium on Programming*, number 1782 in LNCS, pages 1–25. Springer Verlag, March 2000.

[88] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.

[89] Martin Odersky and Matthias Zenger. Scalable component abstractions. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 41–57. ACM, 2005.

[90] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 41–53, 2001.

[91] Kunle Olukotun, Pat Hanrahan, Hassan Chafi, Nathan Grasso Bronson, Arvind Krishna Sujeeth, and Dmitri Makarov. Delite. `http://ppl.stanford.edu/wiki/index.php/Delite`.

[92] OpenMP Architecture Review Board. OpenMP application program interface. Version 3.0, May 2008. available at: `http://openmp.org/wp/openmp-specifications`.

[93] Irina Perfilieva. Normal forms for fuzzy logic functions. *Multiple-Valued Logic, IEEE International Symposium on*, 0:59, 2003.

[94] Quan Phan and Gerda Janssens. Towards region-based memory management for Mercury programs. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 433–435. Springer, 2006.

[95] Quan Phan, Zoltan Somogyi, and Gerda Janssens. Runtime support for region-based memory management in mercury. In Richard Jones and Stephen M. Blackburn, editors, *ISMM*, pages 61–70. ACM, 2008.

[96] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: Design patterns and semantics. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:101–110, 2004.

[97] Filip Pizlo and Jan Vitek. Memory management for real-time Java: State of the art. In *ISORC*, pages 248–254. IEEE Computer Society, 2008.

[98] François Pottier. *Types et contraintes*. Mémoire d'habilitation à diriger des recherches, Université Paris 7, December 2004.

[99] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.

[100] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[101] Vijay A. Saraswat. X10: Concurrent programming for modern architectures. In *APLAS*, page 1, 2007.

[102] Nadathur Satish, Narayanan Sundaram, and Kurt Keutzer. Optimizing the use of GPU memory in applications with large data sets. In *HiPC 2009: International Conference on High Performance Computing*, Kochi (Cochin), India, December 2009.

[103] Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003.

[104] Jens P. Secher and Morten Heine Sørensen. On perfect supercompilation. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 1755 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 1999.

[105] O. Shivers. Control flow analysis in scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174, New York, NY, USA, 1988. ACM.

[106] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In *Proc. FASE*, 2008.

[107] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and gpc. In Donald Sannella, editor, *ESOP*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer, 1994.

[108] S. Doaitse Swierstra. Combinator parsers - from toys to tools. *Electr. Notes Theor. Comput. Sci.*, 41(1), 2000.

[109] Walid Mohamed Taha. *Multistage programming: its theory and applications*. PhD thesis, 1999. Supervisor-Sheard, Tim.

[110] G. Teodoro, R. Sachetto, O. Sertel, M. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira. Coordinating the use of GPU and CPU for improving performance of compute intensive applications slides. In *IEEE International Conference on Cluster Computing*, 2009. `http://homepages.dcc.ufmg.br/~george/publications/2009-cluster.pdf`.

[111] Eli Tilevich, William R. Cook, and Yang Jiao. Explicit batching for distributed objects. In *ICDCS*, pages 543–552. IEEE Computer Society, 2009.

[112] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher Order Symbol. Comput.*, 17(3):245–265, 2004.

[113] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[114] Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

[115] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.

[116] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.

[117] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *Transactions on Programming Languages and Systems*, 29(3):1–16, 2007.

[118] Shinichi Yamagiwa and Koichi Wada. Performance study of interference on gpu and cpu resources with multiple applications. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[119] Tian Zhao, Jason Baker, James Hunt, James Noble, and Jan Vitek. Implicit ownership types for memory management. *Sci. Comput. Program.*, 71(3):213–241, 2008.

[120] Ye Zhou and Edward A. Lee. Causality interfaces for actor networks. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.