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. Impressively 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 \( \text{Rep}[T] \). Here, \( T \) is a normal host language type whereas \( \text{Rep} \) 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 \( \text{Rep} \) types.

\[
\text{def power}(x: \text{Rep}[\text{Double}], n: \text{Int}) = \text{if}(n==0) 1.0 \text{ else } x \times \text{power}(x, n-1)
\]

Instead of concrete implementations of \( \text{Rep} \) 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 \( \text{Rep}[\text{Double}] \) domain. The full proposal shows how to define these operations while keeping considerable syntactic flexibility.

The implementations of all operations on \( \text{Rep} \) types are hidden just like the type \( \text{Rep} \) 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 \( \text{Rep} \) 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 \( \text{Rep} \) types to determine the binding times of variables, which in turn determine when a piece of program is executed: Everything tagged with a \( \text{Rep} \) 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 \( \text{Rep} \) together with common subexpression elimination and knowledge about the associativity of \( \times \), one can automatically rewrite a call such as \( \text{power}(x, 5) \) to the following sequences of instructions:

\[
\text{val } x1 = x \times x; \ \text{val } x2 = x1 \times x; \ \text{x2 } \times x
\]

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
Extended Synopsis

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 \texttt{GPU[T]} for expressions that are executed in a co-processor, whereas \texttt{Local[T]} would mark expressions executed on the main processor. Either type would be produced from a \texttt{Rep[T]} with a domain-specific transformation. The abstract \texttt{GPU} type family would support exactly those operations that can be executed on a GPU, and no others. The concrete implementation of \texttt{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 \texttt{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.
**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, focusing 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-extendible 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 opti-
mization 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 supercom-
pilation. 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 combin-
ing multiple data items into larger packages so that the number of round-trips between communicat-
ing 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 rela-
tive 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


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


<table>
<thead>
<tr>
<th>REFERENCES</th>
<th>Doppler</th>
</tr>
</thead>
</table>
REFERENCES


REFERENCES


