RAW FP: Productivity and Performance through Resource Aware Functional Programming

John Hughes, Mary Sheeran, Koen Claessen and Patrik Jansson

November 24, 2010

1 Main goals

Earlier this year, Swedsoft (an industrial network of large Swedish software developers) called for an order of magnitude improvement in software development productivity over the next decade, if Sweden is to remain competitive. Coincidentally, an improvement of roughly an order of magnitude in productivity over conventional programming languages is the claim that has been made for functional programming since the 1980s. In the early days, that claim was based on the brevity of functional programs—the implementation of QuickSort is two lines in Haskell, but twenty lines in Java—and the well-known observation that developers produce approximately the same number of lines of code per day, regardless of programming language. Today, evidence of improved productivity is much more direct.

Functional programming is seeing rapid growth in industry today. Ericsson were pioneers in the area, designing Erlang around 1990, and using it for the first time in a large product in the mid 1990s—the AXD 301 ATM switch, with around 1,500,000 lines of Erlang code. Writing about the outcome of that project, Wiger estimated a 4 to 10 times productivity improvement over traditional methods [Wiger et al., 2002], supporting the earlier claims. Since then, many other companies worldwide have used the technology to attain business success via higher productivity, notable Swedish examples including BlueTail, sold less than 18 months after start-up for $152 million, and Klarna, which offers invoicing services to web shops and has grown to 350 people in just five years. In recent years Microsoft have adopted functional programming features in their .NET languages C# and Visual Basic, and this year released a functional programming language of their own, F#, as part of Visual Studio 2010. On the JVM, Scala and Clojure offer functional programming with smooth interoperability with Java, and high profile applications have appeared—such as Twitter's back-end processing, which is mostly done in Scala using a concurrency library inspired by Erlang. A major motivation for all this growth is the improved productivity that functional programming brings.

Problem solved? Unfortunately, no—it would be naive to suggest that the Swedish software industry can solve all its problems just by adopting functional programming. Even in the AXD 301, only *part* of the system is programmed in Erlang—the "control plane", which sets up and tears down calls, and responds to protocol requests, is nicely implemented in Erlang, but the "data plane", which actually transmits data streams at gigabytes per second, is programmed in C and C++ to meet extreme performance demands. In the baseband processing of a radio base station, the signal processing is done by C code full of compiler pragmas and tailored exquisitely to the hardware. In ABB's robot control software, and SAAB's avionics, the software must meet hard real-time deadlines. Functional programming languages such as Erlang cannot meet these demands, and so are not yet applicable in these domains.

Does this mean that large parts of the Swedish software industry can never use functional programming? Fortunately, no—there is a way to cut the Gordian knot, and attain the productivity benefits of *writing* functional programs, without incurring the costs of *running* them. The trick is to use functional programming to provide a *domain specific language* (DSL) for the application domain concerned.

1.1 Domain Specific Languages

DSLs have become popular recently [Fowler, 2010], as a way to improve productivity by providing a language tailored to the task in hand, while assuring performance by exploiting domain specific optimizations. Functional programmers advocate domain specific *embedded* languages (DSELs), which provide a DSL via a library in a host language [Hudak, 1996]. This is partly just a shift in perspective—we view the API of the library as the constructs of a DSL—but with a sufficiently expressive host language, and careful design of the API, then the DSLs that result can be very attractive indeed. The great advantage of the approach is that a DSEL inherits the tool chain of its host language, the "look and feel" (so it is familiar to host language developers), and generic features such as modules, interfaces, abstract data types, or higher-order func-

tions. Moreover, the DSEL implementation is very "lightweight"—the DSEL designer can add features just by implementing a new function in the library, and can easily move functionality between the DSEL library and its clients. The ease of experimentation with such a DSEL helps implementors fine tune the design, and enables (some) end-users to customize the implementation with domain specific optimizations.

A DSEL can either implement the DSL's features directly in the host language—or, it can *generate code* from the DSEL program in any other target language, a so-called *domain specific embedded compiler*. In the latter case, the host language is used only at "compile time"; the developer can use the full power of the host language to express the program, but at run-time, only the target language need be executed—for example, Paradise [Augustsson et al., 2008], a DSEL embedded in Haskell, generates Excel user interfaces to C++ algorithms for valuing financial contracts at Credit Suisse. (There is a clear analogy with the "platform independent model" and "model compiler" used in the model-driven development community.)

Of course, achieving this requires careful design of the DSEL, but in return we gain the expressive power of functional programming, while retaining the fine control of resources that is needed in so many industrial applications. We have considerable experience of using Haskell as a host language in this way, to implement DSELs such as Lava (for configuring Field Programmable Gate Arrays [Bjesse et al., 1998]), Obsidian (for programming graphics processors), and Feldspar (for baseband signal processing [Axelsson et al., 2010]).

To illustrate the potential of the approach, in one experiment at Ericsson, Feldspar was used to implement part of the reference signal generation for channel estimation in the LTE Advanced testbed. (LTE, or "Long Term Evolution" is commonly referred to as 4G.) The Feldspar implementation could be structured naturally as a series of cascading processing steps, matching well the domain expert's view of the problem. Not only that, it also generated C code that was 30% faster than the hand-coded reference model, because it permitted optimization (in the form of removal of unnecessary data structures) *between* the steps. It is this effect that we are striving for: high level descriptions that match the thinking of the domain expert, leading to efficient generated code.

Resource aware (RAW) functional programming using DSELs is a broad research agenda, too broad to address in one project. Therefore

The goal of this proposal is to develop domain specific embedded languages for resource-aware functional programming, with associated end-to-end tool-chains, that can deliver an order of magnitude improved productivity in three domains of importance to Swedish industry:

- **signal processing and low-level control in products such as radio base stations (Feldspar),**
- **programming highly parallel heterogeneous architectures containing graphics processors for tasks such as medical image processing (Obsidian),**
- **real-time automotive software built around the AUTOSAR standard.**

These areas have been chosen for the following reasons. The Feldspar component builds on our very promising collaboration with Ericsson, and will require architecture awareness to meet very high performance demands. The heterogenous architecture part builds on our current work with Obsidian, and because it should be useful to anyone with a dual core laptop and a suitable graphics card, offers a possibility of broad impact. The automotive component builds on our collaboration with Quviq AB, and requires us to address the important area of real-time systems. Together, these areas require support for awareness of time, memory, and architecture, and so will force us to solve most of the problems that resource-aware programmers face.

1.2 Verification

Our goal cannot be achieved without reducing *verification costs* as well as development costs, since they typically make up at least half the cost of a software project. Therefore we plan to combine our DSELs with *property-based testing*, a form of automated testing in which individual test cases are replaced by general properties, from which many test cases can be generated. We have developed a property-based testing tool, *QuickCheck*, originally for Haskell, and now commercialised for Erlang by our associated company Quviq. Generating tests from properties improves test coverage at the same time as the volume of test code is reduced, thus improving quality and reducing cost. On a test failure, we can also minimize the test case by searching for similar, smaller cases that also fail, which automates a costly part of fault diagnosis. For example, when QuickCheck rediscovered a bug in gcc's parameter passing, we found from the Bugzilla records that the original discoverers spent two days minimizing the

failing case to the same example that QuickCheck found in minutes—followed by a mere few hours to fix the bug and release a patch.

QuickCheck uses a DSEL for *formal specifications* to express the properties to be tested, which we have extended considerably over the years. The first versions only supported simple properties, such as equations over pure functions, but later versions directly support state-machine specifications, trace properties of asynchronous systems, and atomicity properties to test for race conditions. Such extensions to the formal specification language make properties much easier to write, and are essential for wide adoption of the property-based testing approach. Today, QuickCheck is notching up success stories: at Erlang Solutions, finding faults earlier in an email gateway [Boberg, 2008]; at Gemini Mobile and Basho, testing the NoSQL databases Hibari and Riak for the computing cloud; at Process One, testing asynchronous behaviour in the leading XMPP instant messaging server [Hughes et al., 2010]; at Ericsson, testing the control software of the new LTE base stations. In September, QuickCheck revealed two subtle race conditions that have plagued users of Erlang's bundled database system for years—at least six weeks of fruitless effort has been spent hunting these bugs in the last few years, but QuickCheck enabled them to be found and fixed with just a few days' work. The impact of our work has been considerable—this year the first QuickCheck paper received a *Most Influential Paper* award from ACM SIGPLAN, ten years after publication.

An important lesson from these successes is that a *domain specific property language* is necessary to enable developers to write QuickCheck specifications quickly and easily. We plan therefore to develop such languages for our target domains. We will address both the *expressivity* of properties—how easily we can specify correct behaviour—and the *generation* of good test data for this domain. An important question that arises when test cases are generated, and are never inspected by a person, is *what has actually been tested?* A closely related question is *when have we tested enough?* Such questions are often addressed by coverage measures—but we know that source code coverage is a poor measure of testing quality, more useful for identifying problems than for determining when testing is sufficient. Of more interest is *requirements coverage*—have we tested all the requirements? Yet determining this without manual inspection of the test cases demands that requirements be formalised, which may mean we need a *requirements DSEL* as well. For this to work well, we must ensure that requirements are expressed at an appropriate level of abstraction—if low-level requirements correspond closely to test cases, as is sometimes the case, then the cost of formalising requirements and checking that they have been tested may dominate the cost of formulating and testing properties. If not addressed, this could prevent us reaching the goal of an overall order of magnitude improvement in productivity.

Testing using QuickCheck brings the benefits of formal specifications, without incurring the cost of formal proofs. Indeed, we believe manual—or even machine assisted—formal proofs of software properties to be too costly in our chosen domains. However, *fully automated* formal tools are making strong inroads in industrial scale software analysis, through widely used tools such as Microsoft's Static Driver Verifier, Prefix, and ESC/Java. As we have considerable expertise in building and exploiting proof engines (miniSAT [Eén and Sörensson, 2003], Paradox [Claessen and Sörensson, 2003], Equinox), we will investigate the use of *fully automated* formal verification of carefully selected software components.

Our goals demand both system oriented and disciplinary research on two main themes: resource awareness and verification. In both cases, DSELs will be the approach on which we build, and this will demand disciplinary research on how to design and build DSELs, and system research on how to build and deploy a domain specific tool-chain based on multiple DSELs. While our approach is domain specific, we will also seek *common abstractions* that can be used to ease the development of future DSELs.

2 Description of the project

2.1 Disciplinary research

2.1.1 Resource awareness

Memory awareness. In the kinds of application areas that we target, there is a growing trend towards a high degree of parallelism combined with demands on the programmer to explicitly manage memory and caches. For example, when one programs a GPU in a language like openCL, some memory access patterns lead to high speed, while others lead to memory bank conflicts and have a disastrous effect on performance. Any useful DSL targeting openCL must generate good access patterns where possible. We would like the user to be able to write a program with a structure that matches the problem being solved, but to generate code that is "folded" into a different shape that matches the machine structure, and in particular exploits memory in an efficient manner. We will develop combinators (or higher order functions) to capture commonly used memory access patterns, and to enable both optimization of and easy experimentation with memory use.

The problems multiply when the available memory close to computational units is severely limited, as happens in some telecoms platforms and even to

some extent in GPUs. In that case, it becomes necessary to overlay data, swapping parts of data structures in and out as they are needed. Here, again, we propose a DSL to capture the necessary data movements, and a type-based technique to determine dynamic memory use, enabling the avoidance of dangling pointers. It will be necessary to give the programmer feedback on memory use, cache behaviour and compliance with memory bounds, and this will entail the development of a suitable cost model that captures both space and time use (see below). This demands also that the architectural models should capture memories and caches at a sufficiently fine level of detail.

Time awareness. In all of our chosen domains, the programmer must be aware of a notion of time, and must reason about the time behaviour of his programs. The exact nature of the requirements on time behaviour varies between applications. In graphics processing on GPUs for realistic image rendering, sheer speed is what drives the programmer. In radio processing, different standards have different granularities of time, and how hard the demands are may depend on whether one is programming the uplink or the downlink. Typically, one is aiming to service many users simultaneously, so the computations must not only be timely but also efficient in the use of other resources like memory or communication. Failure to meet deadlines may cause the link to go silent very briefly, or the dropping of calls. In the automotive industry, static scheduling is used to guarantee completion of tasks in a cycle, and the AUTOSAR standard aims to provide components that can be combined in ways that give the required timing behaviour. Our aim here is to consider ways to add real time to a simple functional DSL and to experiment with ways to specify and control timing behaviour. We will develop ways to *test* programs for compliance with timing specifications, and this in turn demands the creation of a DSL for the specification of performance requirements.

Architecture awareness. High-performance implementations often require architecture-aware compilation: we need to model computer architectures and their memory hierarchies, taking into account costs of communicating data across and between levels of the hierarchy, and costs of accessing data in particular locations with particular access patterns. As we have long experience of hardware modelling in functional languages, we will extend that work to architectures. If we think of compilation as reshaping the program to fit the architecture, then information about the architecture must flow upwards and guide those program transformations. We see an opportunity for yet another DSL here, to capture the necessary information. We plan to collaborate in the development of this DSL and its use in architecture-aware compilation with Sally A. McKee (a computer architect who is an expert on memory hierarchies and an author of the classic “memory wall” paper [Wulf and McKee, 1995]).

Cost models and analyses. If the programmer is to gain fine control of resource use (memory, time, communication, exploitation of architecture), it will be necessary to develop cost models that allow for analyses of his program to provide early feedback about how near or far he is from his goals. Simply looking at the generated code, or running it and measuring some of these properties is not enough. One needs to also give feedback that is related to the structures and types in the user program, We expect to perform these analyses by various forms of abstract interpretation. (Note that in the system-oriented research, we will need to develop a good understanding, in each domain, of how best the programmer should work, refining his implementation towards one that meets his functionality and performance goals. There, we expect both cost models and more standard profiling of actual runs to play an important role.)

Optimization by search. We will investigate the use of search to optimize the choice of a combination of program sub-parts that performs well on the given architecture. Work on Spiral [Püschel et al., 2005], a DSL and tool for auto-generation of platform-tuned libraries for DSP and other transforms was a major inspiration at the beginning of the Feldspar project. Our data-flow style algorithmic descriptions and use of combinators to capture algorithmic structure should permit a very similar approach to the tuning of transform implementations. Working directly in Haskell, we have shown that the use of lazy dynamic programming enables the discovery of parallel prefix (or scan) networks that improve on current best known networks [Sheeran, 2010]. In the above examples, one is exploiting the fact that the algorithms being implemented have a great variety of possible implementations, but all built using a relatively simple top level structure. The challenge in this project will be to generalise the approach to cover a broader range of Feldspar or other RAW programs.

Robustness. Handling errors is the bane of software development: in many systems, error handling code makes up $\frac{2}{3}$ of the code, accounts for more than $\frac{2}{3}$ of the bugs (because it is often poorly tested), and is responsible for $\frac{2}{3}$ of system crashes. Erlang’s “let it crash” philosophy reduces error handling code drastically, by isolating failures to single processes and recovering at a higher-level using a supervision tree structure. We plan to borrow the idea for our

DSELs, but it does require language support in the form of processes and links. We will investigate ways to add these features to a DSEL, and generate efficient implementations. This will result in a strongly-typed Erlang-like language, with compilation to C.

2.1.2 Verification

We plan to develop verification tools for our DSELs, and also to *verify our own tools* using property-based testing. This will require disciplinary research on a variety of topics.

Domain specific specifications. We plan to complement each DSEL with a domain specific specification language, making important properties of the code easy to specify and test. The three challenges are *expressing what it means to be correct*, generating *good test data*, and finding *powerful shrinking strategies* for minimizing failing test cases.

Correctness is often surprisingly hard to specify, at least without reimplementing the software under test (which defeats the purpose since the reimplementation is likely to suffer the same bugs as the original). However, the task can be eased by finding reusable abstractions, such as finite state machines and the "temporal relations" we have developed to specify asynchronous systems [Hughes et al., 2010]. We plan to develop additional abstractions motivated by our chosen application domains. For example, we have as yet no good approach to *mocking* in property-based testing—the replacement of a component which is called by the software under test by a mock version which makes valid responses to the calls generated by the test. In a random test we do not know what these calls will be, making mocking harder. As a result we can easily test a *server* with a random client, but it is much harder to test a *client* with a random server. Finding a good approach to property-based mocking is one of our goals.

We generate test cases dynamically and at random, which enables us to continue running new tests long after all reasonable coverage criteria have been fulfilled—and it is surprising how often one of these "superfluous" tests (by coverage criteria) reveals a subtle bug. However, we are always careful to control the distribution of generated data, so as to test efficiently. Good test data is test data that reveals faults fast—but what characterizes good test data varies from domain to domain. For example, for finite state machines we distribute testing effort as evenly as possible across all the transitions, which requires weighting transitions at each state to direct testing towards hard-to-reach parts of the state space. Determining these weights is a difficult optimization problem. This situation arises frequently: *local* random choices must achieve a good *global* distribution, which makes the assignment of probabilities tricky.

Another challenge is satisfying complex invariants on the generated data, because random choices at one point may restrict the choices available at another point, perhaps in unfortunate ways. For example, when generating random *programs* to test compilers (or to test our DSELs!), if variable and function types are chosen independently, then most variables will never be used—because no function accepts an argument of the right type. By studying the test data generation problems in our application domains, we hope to develop a systematic approach to these problems.

Shrinking failing tests to minimal examples is critical to QuickCheck's usefulness. But shrinking strategies differ from domain to domain. Simple strategies—replacing numbers by smaller ones, or replacing data structures by sub-structures—are quite widely applicable, but need to be supplemented by domain specific ones. For example, to minimize failing tests of a soft real-time system, we found we had to shrink API calls to sleeps, which would make no sense in other contexts. Shrinking data while preserving invariants can also be challenging: shrinking one part of the test may make another part invalid, or may require it to be shrunk correspondingly. Broken invariants can be fixed in a variety of ways: we plan to investigate systematic approaches to this problem based on domain specific shrinking in our application domains.

Non-functional properties. In a RAW programming language, resource properties such as time and space bounds, or memory access patterns, are important. In some cases these may be verified statically (our own work on *sized types* [Hughes et al., 1996] has been applied to this problem [Hammond and Michaelson, 2003]). However, we also plan to check resource properties by testing.

Test quality assessment. At present, it is hard to assess the quality of QuickCheck testing, or to determine how many tests to run—whether 10,000 tests are enough, or 100,000 tests, or 1,000,000 tests. High source code coverage is too easy to achieve to be really useful. We will investigate more appropriate coverage measures, especially with respect to coverage of requirements, perhaps resulting in a requirements DSEL whose purpose is to assess the quality of testing. One useful approach to measuring test quality is *mutation testing*: once all tests pass, then one generates a large number of "mutants" (i.e. small variations that simulate faults) of the software under test, and measures the proportion of mutants that are "killed" by the test suite—i.e. fail at least one test. The more mutants killed, the better the test suite. We will integrate mutation testing into our DSELs—since our DSELs already

generate code, then it will not be difficult to generate mutated code also.

Specification validation. Like programs, specifications are often wrong when first written. In practice, QuickCheck specifications are debugged by testing them against the code they specify. Failing tests reveal an *inconsistency* between the specification and the implementation, but the error may lie in either one. This approach works well in an agile development process, where properties and the code they specify are developed together, but is problematic if the specification is developed *first*. To support this kind of development, we will investigate ways to analyse and debug such a specification *before* the corresponding code is written.

Mining specifications. Property-based testing replaces many individual test cases by a few general properties. Yet experiments show that some developers find test cases—which are really examples—easier to write than general properties [Claessen et al., 2010a]. Moreover, the lack of a QuickCheck specification for legacy code can make developers reluctant to write properties to test new functionality. It is tempting simply to add a few test cases for the new functionality, even though this risks failing to discover bugs caused by interactions between the old and the new [Rivas et al., 2010]. Both observations motivate the development of methods to *mine specifications* from existing code. We have already developed a prototype tool called *QuickSpec* which can discover equational specifications of pure functions by automated testing [Claessen et al., 2010b]; we plan to extend it to handle a much wider class of specifications and programs.

Testing concurrency. Concurrent programs are bedevilled by race conditions, which require exploring alternative schedules during testing. Concurrent behaviour is also, in general, hard to specify. We have found *serializability* to be a useful *generic* property that many concurrent APIs ought to satisfy, and used it to find race conditions in industrial code [Claessen et al., 2009]. We are now extending this work to test random schedules that are particularly likely to fail. A particularly important problem is the automated *simplification* of a failing test when a race condition is detected—we must simplify both the test case, and the schedule, together. We plan to develop these methods further, and apply them to the DSELs.

Exploiting proof engines. At Intel, there has been a recent breakthrough in formal firmware verification using SMT solvers [Franzen et al., 2010]. We see opportunities for fully automated formal analysis of some carefully chosen algorithmic software components. With our strong background in formal verification [Sheeran et al., 2000], model checking, SAT solving [Eén and Sörensson, 2003] and first order model finding and theorem proving (Paradox [Claessen and Sörensson, 2003], Equinox), our group is uniquely qualified to develop new forms of formal software verification. The fact that we are working in strictly restricted domains and have a very simple, purely functional intermediate language greatly increases the chances of success in using automated formal verification. It may be possible to significantly boost verification efficiency by replacing a huge number of tests with one call to a proof engine. However, the choice of when to use these engines is not trivial; for example, verifying a program performing a computation on an array of any length might be infeasible, but choosing a specific upper bound on the size of the array might make the problem tractable. In this sense, we might have to pick some of the inputs to the program as concrete values (as we do in testing), while other inputs are verified symbolically. Our DSLs typically allow such *symbolic evaluation* to take place, and this in turn can assist the verification by providing additional information about the structure of the program. We plan to identify as many such "exploitable corners" in our verification problems as possible.

2.1.3 DSEL framework

Building a DSL as an embedded language has many advantages, in particular the reuse of features of the host language (syntax, type system, module system, compiler, debugger, libraries, testing tools, etc.) However, many embedded DSLs have more in common than just their host language. We have identified many questions that come up again and again as we build DSELs. We cannot list them all here, but choose a small number of important topics and indicate the kinds of question that arise in those areas:

Comprehensible Error messages A known weakness of embedded DSLs is that the user can be faced with baffling error messages because the embedded compiler does not have access to the original names of functions and variables (and their locations) of the user program. The user wishes to debug his program at the level of his source code, and not in terms of the generated code, so there is a strong analogy with source level debugging. Can a generic solution to maintaining a link between generated code and source program be developed and implemented as a library?

How to represent and manipulate parallel arrays DSLs often capture data parallelism using a special parallel array data structure. Our work on Obsidian (for GPU programming) and on Feldspar has brought home to us the importance of choosing an

appropriate representation that gives the user both a simple high level interface and access to the computation patterns that he would like to attain in the generated code. We are currently prototyping a new form of array that separates the concepts of pushing to and pulling from arrays, giving greater expressiveness. Further work on this will influence the design of Feldspar and Obsidian, probably leading to a convergence. When we started work on Feldspar, we were unaware of the importance of this research topic, but we now feel that the question of how to represent arrays is central to resource aware programming.

Simplification of generated code Generated code is often unnecessarily complicated. For example, a program such as "if (0 = 1) then p else q" can be simplified to just "q". Many back-ends need a simplification pass like this, but relying on a separate compiler for the back-end is often not sufficient, because it is important to exploit invariants guaranteed by the DSL.

We will perform a systematic study to document, generalize and implement key components that are common to many DSELs. We will build a DSEL framework that will allow the DSEL builder to combine generic building blocks for large parts of the implementation, leaving him to concentrate on the question of what exactly the embedded DSL should express. Our aims are similar to those of *Language Workbenches* [Fowler, 2010].

2.2 Demonstrator

The demonstrator will consist of complete tool-chains for each of the three chosen application areas. Each tool-chain will support formulation of requirements, design, implementation and debugging of the code, and verification through testing and, where appropriate, automated proof. By adopting this end-to-end approach we strive to improve productivity throughout the software development process, not just in writing the code.

The Feldspar DSEL targets multicore DSP processors, and is intended to express both the signal processing algorithms used in base stations, and the coordination of the algorithmic blocks. Today's version of Feldspar can express signal processing algorithms elegantly, and generate ANSI C implementations, but the demonstrator will also apply architecture-specific optimisations using a separate architectural model, and will support the coordination layer. The coordination must be achieved using limited memory and meeting real-time deadlines, and is the motivating application for our work on memory-aware Erlang-like DSELs discussed above. Since the Feldspar work is already quite advanced, then we plan to complete this part of the demonstrator within two years after the start of the project, giving us an opportunity to apply the lessons learned from it to the other two.

The Obsidian DSEL targets manycore systems with one or many graphics processors, and is intended to support data-parallel and cache aware programming for applications such as medical image processing and on- or off-line image rendering. Today's version of Obsidian allows the programmer to specify single so-called "kernels" of computation more concisely and at a higher level than, for example, NVIDIA's CUDA language, but is somewhat restricted in the kernels it can express. The demonstrator will be much more expressive, and will also support coordination between the kernels to implement much larger and more complex highly parallel computations. The key to high performance (either in CUDA or in Obsidian) lies in the memory access patterns used. Apparently similar programs with slightly different access patterns can exhibit widely different performance. Yet because these patterns are *implicit*, then optimizing them is something of a black art. The demonstrator will support such patterns *explicitly*, enabling programmers to adapt their code easily in this respect. This is a prime motivation for the work on memory awareness above.

The automotive DSEL targets the electronic control units (ECUs) which run the embedded software in vehicles, and is intended for programming AUTOSAR components. The AUTOSAR standard specifies a component-based architecture, with detailed requirements on each component, but it is proving difficult to establish that component implementations actually conform to the standard. Quviq is developing open QuickCheck specifications to address this problem; the demonstrator will combine these specifications with a DSEL for *implementing* AUTOSAR components, with high confidence in their conformance. Since AUTOSAR is an evolving standard, there will be a continuing need to update specifications and implementations, motivating the development of a DSEL to ease this task. AUTOSAR components are real-time systems which must meet important timing requirements, motivating our work on *time awareness* described above. One of the problems of a component-based approach is that standardised interfaces between components can impose a run-time and memory cost, that bespoke software for the same task would not encounter. To mitigate this, AUTOSAR allows components to be clustered, so that the cluster provides the standardised interface at its borders, but internally may be implemented more efficiently. One of the goals of our AUTOSAR DSEL is to support component *fusion*, so that this clustering can be achieved automatically.

2.3 System research

Our way of working is case study driven. We first build prototype tools and try them on industrial case studies. This in turn places new demands on the tools, demanding new disciplinary research, which feeds the next version of the tool and the next round of experiments.

2.3.1 years 1 and 2

Mobile broadband. In Mobile broadband, our most mature domain, system research in the first two years of the project will concern the deployment of the Feldspar tool, including support and observation of real users. This will give vital feedback on usability, and will likely lead to significant changes in the tool itself. We will need to develop a detailed understanding of what the user needs in the form of early analyses and profiling. Real case studies will drive this work, and it is important to have access to real users, to try to ensure that a higher level of programming still gives access to code of comparable performance to that produced in the current development process. We will take advantage of the fact that we have one captive real user in the form of A. Persson (industrial doctoral student and baseband expert). Initial experiments with the use of Feldspar by Ericsson research engineers at Baseband Research were simultaneously very promising and very revealing of issues that need to be addressed. We hope to continue this collaboration, in particular, aiming to *measure* productivity improvements, since this is, after all, our ultimate goal.

Expanding into new domains. In the first six months of the project, we will hold workshops with interested companies in the automotive and image and media processing areas. Our aim will be to define the specific properties of these domains, and to attract industrial collaborators who are willing to specify their requirements on a tool chain, so that our prototypes can be made to address the right problems.

2.3.2 years 3 to 5

Mobile broadband. The initial phase of the project will have produced a complete tool chain for low level DSP plus the lowest levels of control. We aim to study its effectiveness in real use, provided Ericsson (or another company) are interested in taking part at that time. This study will give valuable insight into the effects of introducing new programming language technology to users whose background is in low level C or C++ programming. It will guide us in our attempts both to broaden the application of the Feldspar tool chain, and to apply similar techniques in other domains.

Simultaneously, we will begin to analyse of needs of the next layer of the software stack by performing initial case studies and developing prototype extensions to the tool chain.

Automotive and/or image processing We will choose one or both of these domains and apply a similar tool development process to that used in the case of Feldspar in the previous phase: case studies, observation and measurement feeding the disciplinary research and the development of the two corresponding DSELs.

3 Competitiveness

The Swedsoft report calls for a ten times improvement in the efficiency of software development by 2020, and the foundation's call focusses on methods that will transform the development of software intensive systems. The goals of this project are to develop methods of precisely this sort.

We are targeting specific application domains of great national importance:

- Infrastructure for mobile broadband, the first priority area named in the foundation s call for proposals, and the core business of Ericsson, the world s fifth largest software developer. Our work in this area is actively supported (and partially funded) by Ericsson.
- Automotive software, the second priority area named in the call, and of critical importance to Volvo, Saab, and many of their suppliers 25-35% of the value of a vehicle now consists of software.

Success in these domains has the potential to improve competitiveness dramatically in a substantial part of the Swedish software industry.

4 Milestones (what and when)

Two years:

- A complete tool chain for digital signal processing (based on current Feldspar) including preliminary industrial case studies.
- A prototype tool chain for part of the automotive application area.
- A prototype tool chain for functional many-core graphics processor programming.

Five years:

- Broadened applicability (within the software stack) for the future Feldspar tool chain
- Real case studies of the application of Feldspar technology within Ericsson
- A complete tool chain for the Automotive DSL

- A complete tool chain for many-core graphics processor programming.
- A general framework and design principles for DSL development (making it applicable for new domains)

5 Research group and forms for cooperation

The work in this proposal will be carried out in the Functional Programming Group at Chalmers. The group consists of four senior researchers (the applicants, three of whom are professors), one research assistant, one industrial research assistant (IFA), three postdocs, and eight doctoral students. The four applicants have more than 10,000 citations in total, according to Google Scholar. John Hughes is the project leader, with Mary Sheeran as co-leader. Because John Hughes is 50% employed at Quviq, co-leadership is necessary. In addition, this arrangement ensures that work on testing and on resource aware DSLs will be well integrated.

The applicants have a long track record of working together successfully: however, John Hughes will take primary responsibility for the automotive DSEL, Mary Sheeran will be responsible for Feldspar, Koen Claessen for Obsidian, and Patrik Jansson for the DSEL framework.

In 2010 the group was funded by VR (5.6 MSEK), SSF (Mobility award for Sheeran, 0.96 MSEK), EU FP7 (mainly ProTest, about 1.4 MSEK) and Ericsson (Feldspar project, 1.5 MSEK)—about 9.5 MSEK of external funding in total. SSF funding for this project would increase the group's annual funding by around 4 MSEK, taking into account projects which are ending. This represents substantial, but manageable growth.

The funding requested will cover 25% of each of the applicants' salaries, two research assistants, one postdoc, and four doctoral students. This translates to one senior person and one student for each DSEL, and for the DSEL framework, averaged over the project. The intention is to recruit the two research assistants at the beginning of the project (possibly from among our existing postdocs), together with two new doctoral students. The postdoc funding will partly be used to fund shorter term visitors. The remaining funding for two doctoral students will initially be used to support students already working in the group, then to recruit new students as those students complete their doctorates.

Ericsson has promised an additional 166kSEK for work on Feldspar in 2011 should we be successful in obtaining funding of this proposal.

We have long experience in DSEL design and in property-based testing, as described above. Through Quviq, we enjoy insights into the problems of applying property-based testing in industry. We have access to expertise in baseband signal processing through our collaboration with Ericsson, and also through Anders Persson, an industrial PhD student in our group, who has 8 years of experience in this area at Ericsson. We plan to collaborate with Sally A. McKee (Chalmers) to exploit the memory hierarchy, and with Ulf Assarsson (Chalmers) in the area of image processing and graphics processor programming. Automotive software is new to us, but Quviq is testing AUTOSAR components with QuickCheck, and has a strong interest in the area, and the new Software Centre at Chalmers will involve automotive companies. We plan to gain expertise in real-time systems through our colleagues in the Dependable Real Time Systems Group (who have long worked with the automotive industry, and recently started an AUTOSAR-related project) and through Johan Nordlander (Luleå University of Technology), who is visiting both our groups. In the area of functional programming in general, and of domain specific embedded compilers, we collaborate with Simon Peyton Jones and Satnam Singh at Microsoft Research in Cambridge, where Koen Claessen is currently on a three-month secondment. Work with Colin Runciman from York (an expert on profiling of functional programs) is planned, funded by the Ericsson Foundation.

The RAW FP project will be governed by a steering group with representatives from Swedish industry (from Ericsson, the automotive industry and the Swedsoft community) and the international academic community.

We have good experience of researchers (both seniors, postdocs and PhD students) working part time on industrial projects and we have seen clear benefits of actually sitting part time at Ericsson. In the RAW FP project we want to continue and broaden this industrial collaboration with possibilities for postdocs and doctoral students to spend some time working for Quviq and other associated companies (in the Automotive and Image Processing areas).

6 Handling of IPR issues

The IPR produced in the project will be the property of the individual researchers, according to the Swedish *lärarundantag*, except for research directly funded by Ericsson, which belongs to Ericsson. At present, Ericsson is funding the development of Feldspar and the results of that project are owned by Ericsson. At Ericsson's initiative, the Feldspar implementation is released as open source software under the BSD3 licence. Hitherto Feldspar has been released by Ericsson (and not by the academic partners in the project).

Our intention is to release the *implementations* of all three DSELs as open source software, under a

BSD3 licence to facilitate commercial use. (Experience suggests that programming languages, even innovative ones, are difficult to sell as products). Some parts of the surrounding tools may however be commercialised via Quviq or another spin-off instead. These decisions will be made by agreement between the company and the researchers concerned at the relevant time.

7 Industry references

"Ericsson has successful ongoing collaboration with the Functional Programming group at Chalmers on the topics of property based testing and Domain Specific Languages. The research content of this proposal is in line with our research needs and we look forward to further collaboration with the group."

Anders Caspár, Director Ericsson SW Research, Ericsson AB, Färögatan 6, SE-164 80 Stockholm. (Phone) +46 10 715 3844. (eMail) `anders.caspar@ericsson.com`

"Ericsson is a world leader in the production of mobile radio products. Software development is by far the biggest part of product development in this area. In order to keep our competitive edge, we must continuously work with and evaluate state of the art software technology. Our low level, high performance, embedded baseband software must be tailored closely to the hardware on which it runs, particularly with regard to the memory hierarchy. Yet we would like to raise the level of abstraction at which it is developed, in order to increase programmer productivity and ease verification. Thus, the research goals in this proposal are closely aligned to those of Ericsson's Development Unit Radio (DURA), and are therefore of strategic importance for the company."

Mike Williams, Ericsson AB, Dept FJT/W, DURA Systems and Technology, SE-164 80 Stockholm. (Phone) +46 10 717 1855 (eMail) `michael.williams@ericsson.com`.

References

L. Augustsson, H. Mansell, and G. Sittampalam. Paradise: a two-stage DSL embedded in haskell. In *ICFP '08: Int. Conf. on Functional programming*, pages 225–228. ACM, 2008.

E. Axelsson, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A Domain Specific Language for Digital Signal Processing algorithms. In *Proc. Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MemoCode*. IEEE Computer Society, 2010.

P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Int. Conf. on Functional Programming, ICFP*, pages 174–184. ACM, 1998.

J. Boberg. Early fault detection with model-based testing. In *Proc. 7th ACM SIGPLAN workshop on Erlang*, pages 9–20, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4.

K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.

K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. *SIGPLAN Not.*, 44, August 2009.

K. Claessen, J. Hughes, M. Pałka, N. Smallbone, and H. Svensson. Ranking programs using black box testing. In *Proc. 5th Workshop on Automation of Software Test*, AST '10, pages 103–110, New York, NY, USA, 2010a. ACM.

K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing formal specifications using testing. In G. Fraser and A. Gargantini, editors, *Tests and Proofs*, volume 6143 of *LNCS*, chapter 3. Springer-Verlag, Berlin, Heidelberg, 2010b.

N. Eén and N. Sörensson. An extensible SAT-solver. In *The SAT Conference*, 2003.

M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

A. Franzen, A. Cimatti, A. Nadel, R. Sebastiani, and J. Shalev. Applying SMT in Symbolic Execution of Microcode. In *Proc. Formal Methods in Computer Aided Design (FMCAD)*, 2010.

K. Hammond and G. Michaelson. Hume: a domain-specific language for real-time embedded systems. In *Proc. 2nd int. conf. on Generative programming and component engineering*. Springer, 2003.

P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28, December 1996. ISSN 0360-0300.

J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proc. 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 410–423, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.

J. Hughes, U. Norell, and J. Sautret. Using temporal relations to specify and test an instant messaging server. In *Proc. 5th Workshop on Automation of Software Test*, AST '10, pages 95–102, New York, NY, USA, 2010. ACM.

M. Püschel et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

S. Rivas, M. A. Francisco, and V. M. Gulías. Property driven development in Erlang, by example. In *Proc. 5th Workshop on Automation of Software Test*, AST '10, New York, NY, USA, 2010. ACM.

M. Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *Journal of Functional Programming*, (accepted for publication, in press), 2010.

M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. Int. Conf. on Formal Methods in Computer Aided Design*, volume 3312 of *LNCS*. Springer Verlag, 2000.

U. Wiger, G. Ask, and K. Boortz. World-class product certification using Erlang. In *ERLANG '02: Proc. of the 2002 ACM SIGPLAN workshop on Erlang*, pages 24–33. ACM, 2002.

W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23, March 1995.

Other costs

The project starts in September 2011 and runs for five years. The total budget of 33M SEK is almost exclusively used to partially fund 11 researchers ranging from seniors to PhD students. The other costs (525 kSEK/year for five years) come from travel and computer equipment:

Travel: We will be active in international research events like conferences, workshops and developers' meetings. We base our estimate (440 kSEK/y) on current travelling within similar projects.

Equipment: The development of advanced software prototypes requires good personal computer equipment and access to the hardware platforms we target (both multi-core and many-core). We calculate an average of 23k SEK / year / computer and a life-time of three years resulting in a total of 85k SEK / year for the whole project.