Language support for the construction of high performance code generators

Georg Ofenbeck[†] Tiark Rompf [*‡] Alen Stojanov[†] Martin Odersky[*] Markus Püschel[†]

[†]Dept. of Computer Science, ETH Zurich: {ofgeorg, astojanov, pueschel}@inf.ethz.ch
[‡]Oracle Labs: {first.last}@oracle.com [*]EPFL: {first.last}@epfl.ch

1. Introduction

The development of highest performance code on modern processors is extremely difficult due to deep memory hierarchies, vector instructions, multiple cores, and inherent limitations of compilers. The problem is particularly noticeable for library functions of mathematical nature (e.g., BLAS, FFT, filters, Viterbi decoders) that are performance-critical in areas such as multimedia processing, computer vision, graphics, machine learning, or scientific computing. Experience shows that a straightforward implementation often underperforms by one or two orders of magnitude compared to highly tuned code. The latter is often highly specialized to a platform which makes porting very costly. One appealing solution to the problem of optimizing and porting libraries are program generators that automatically produce highest performance libraries for a given platform from a high level description. Building such a generator is difficult, which is the reason that only very few exist to date. The difficulty comes from both the problem of designing an extensible approach to perform all the optimizations the compiler is unable to do and the actual implementation of the generator.

To tackle the latter problem we asked the question: *Which programming languages features support the development of generators for performance libraries?* We investigated that question through a case study [2], in which we implemented a subset of the Spiral program generator [3] in the host language Scala. This paper summarizes the key novelties we explored throughout the referenced paper.

2. Language support in Scala with LMS

Throughout the case study we map a list of requirements extracted from several well established program generators and map them to language constructs in our chosen target environment: Scala with Lightweight Modular Staging [4]. The list of requirements consists of:

- *Describe problem and algorithmic knowledge through one or multiple levels of DSLs.* Program generators need to model problems and algorithms at a high level of abstraction and may need to optimize code at multiple intermediate abstraction levels.
- *Specify certain optimizations and algorithmic choices as rewrite rules on DSL programs.* DSLs can be used for high-level optimization through rewriting (e.g., parallelization in Spiral) but rewrite rules can also be used to express algorithmic choices. Doing so facilitates empirical search ("autotuning"), which in many cases is required to achieve optimal performance.
- *Design high-level data structures that can be parametrized to generate multiple low-level representations.* Often generated libraries need to support multiple input/output data formats. A common example is interleaved or split or C99 format for complex vectors, meaning there will be one library function per format. A generator should be able to abstract over this choice of low-level data formats to ensure maximal compatibility [1].
- *Rely on common infrastructure for recurring low-level transformations.* There are certain transformations common in high performance code that are necessary but particularly unpleasant to implement and maintain manually. Examples include a) unrolling with scalar replacement, b) selective precomputation during code production or initialization, and c) specialization (e.g., to a partially known input). Since these transformations are so common, they should be implemented in a portable way using suitable language features.

2.1 Algorithmic Knowledge as Multiple Levels of DSLs

We implement DSLs using the infrastructure provided by LMS, described in detail in [5]. To produce a regular program from our metaprogram, we would translate a DSL by emitting a function for each node in the form:

```
def f(in: T): T
```

The resulting functions we would chain during translation to yield our final program by calling the topmost element of the chain. To facilitate multiple levels of DSLs we extend the common use case of staged DSLs, by changing the output of our meta program to be a meta program by itself. To implement this we simply add staging annotation to the emitted functions in the form of

```
def f(in: Rep[T]): Rep[T]
```

where `Rep[T]` is the common staging annotation used in LMS. Note that `Rep[_]` is a higher order type.

2.2 Optimization through DSL Rewriting

The implementation of DSL rewrites benefits heavily from the build-in patter matching support of Scala. Arithmetic simplification e.g. can be expressed with the following code

```
def transformStm(stm: Stm):Exp[Any]=
 stm.rhs match {
  case NumericTimes(a, b) => (this(a), this(b)) match {
    case (_, Const(0)) | (Const(0), _) => Const(0)
  }
  case _ => super.transformStm(stm) }
```

2.3 Abstracting Over Data Representations and Code Style

Staging annotations in LMS are done via types, which in turn enables us to use the type system in surprising ways. `Rep[_]` is a higher order type, which once parametrized with a concrete type `T` becomes the concrete type `Rep[T]`. Therefore we can translate a DSL into a function graph, consisting of elements which are parametrized in type `T` as follows:

```
  def f[T](in: T): T
```

With this construct, the function graph can be called with a concrete type `T` or with its staged alternative `Rep[T]`. This allows us to effectively turn staging on or off.

2.3.1 Selective Precomputation

To activate precomputation selectively we utilize the mechanism described above as sketched in the snippet below:

```
val input: Rep[Double]
val index: Int
val result1 = input * f(index: Int) //Precompute
val result2 = input * f(index: Rep[Int]) //stage
```

Note that the only difference between the two result variants is the type of the input. The implementation of the selectively precomputed function can be done completely abstract and only a single code implementation is required for it.

2.3.2 Abstraction over Code Style

Putting staging expression on different positions in the meta program yields fundamentally different code in the staged program. Putting the staging annotation around only primitiv types yields code that is scalarized and unrolled due to the fact that every array access and the loop evaluation are done at staging time and will not appear in the staged program.

```
  def f(in: Array[Rep[Double]]): Array[Rep[Double]] =
        for (i <- (0 until n):Range) ...
```

Putting the staging annotation around the array will also yield unrolled code, but every array access and store is now part of the staged program.

```
  def f(in: Rep[Array[Double]]): Rep[Array[Double]]
        for (i <- (0 until n):Range) ...
```

Finally, putting the annotation also at the loop construct will lift the loop itself into the staged program, yielding looped code that works on arrays.

```
  def f(in: Rep[Array[Double]]): Rep[Array[Double]]
        for (i <- (0 until n):Rep[Range]) ...
```

Combining these observations with the selective staging utilized for precomputation we can now write functions of the form:

```
  def f[A[_],L[_],T](in: A[Array[T]]): A[Array[T]] =
        for (i <- (0 until n):L[Range]) ...
```

This construct allows us to abstract all the codestyles mentioned and choose the code style by type parametrization of the function. For example:

```
val x  = f[Rep,Rep,Double](y)
```

will yield the looped code working on arrays and

```
type NoRep[T] = T
val x  = f[NoRep,NoRep,Rep[Double]](y)
```

will yield the unrolled, scalarized code. Note that functions which are ill-parametrized will yield a type error at compile time. We consider this a major benefit compared to other metaprogramming approaches such as e.g. using C++ templates to achieve the same goal. In such an approach errors would by default only manifest once the concrete implementation is generated.

2.3.3 Abstraction over Data Representations

Finally we combine all of the above with standard object oriented programming to abstract over different data layouts. We define a base type Vector

```
abstract class Vector[A[_], L[_], T] {
  def apply(i: A[Int]): T
  def update(i: A[Int], y: T)
}
```

which we then extend with subtypes that implement concrete implementation e.g.

```
class Staged_Unrolled_List
  extends class Vector[Rep[_], NoRep[_], T] {
  def apply(i: Rep[Int]): T
  //call to staged list apply
  def update(i: Rep[Int], y: T) }
  //call to staged list update
```

This object model allows us to pass only the super type through the generated call graph and therefore construct a solution that is abstract in the data layout. The concrete instantiation can be selected again by passing the according type into the top of the generated callgraph. In the full paper we show a more involved implementation which enables us to abstract over all different types of complex data (C99, Interleaved Complex etc.) which is build around the same principle.

3. Conclusions

Ever since the term multi-stage-programming was coined by by Taha and Sheard [6] it was related to program generation. In our case study we demonstrated that multi-stage-programming, which is done trough types, enables us to couple it with modern program language constructs such as generics. Through this combination we are able to build powerful techniques for program generations such as abstract over precompuation, abstraction over the code style of the resulting program and finally the data layout of the target program.

References

[1] B. Aktemur, Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shonan challenge for generative programming: short position paper. In *Proc. Partial evaluation and program manipulation (PEPM)*, pages 147–154, 2013.

[2] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in scala: Towards the systematic construction of generators for performance libraries. In *International Conference on Generative Programming: Concepts & Experiences (GPCE)*, 2013.

[3] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[4] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[5] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented dsls. ., (arXiv:1109.0778), Sep 2011. Comments: In Proceedings DSL 2011, arXiv:1109.0323.

[6] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.