General Automatic Stage Separation with Applications to Algorithm Derivation

Nicolas Feltman et al.

December 10, 2013

1 Introduction

Often when programming high-performance systems, a programmer will wish to specialize a bivariate function to one of its inputs. This need may arise because one input varies at a higher frequency than the other, and we wish to avoid repeated calculation. It may also be that one input is available at a time when computational resources are less expensive. It's even possible that knowledge of an input could allow further code optimizations that would have otherwise been unwise to speculate. Broadly, the solutions to this specialization problem can be called *staging*, in that different parts of the code are intended to run at different points in time, *i.e.* stages. The general theme is to do some work with the *immediate input*, producing an intermediate result, and then finish the rest of the computation once the *delayed input* is available. In particular, these staging techniques include:

- **Program Generation** A common solution is to generate code in some target language that itself will take in the second stage input. Using the equational notation of [Jones 96], we say that a programmer defines *genf* in a language C so that

  $$[[genf]_C\ x]_T\ y = [f]_L\ (x, y)$$

  for all x and y. Examples of this technique include most compilers (where f is a hypothetical interpretter, and the first stage input is the source program), or even specialized compilers like Spiral (where the first stage is simply the dimension of the DFT). The benefit of this technique is full control, and the fact that implementations of the target language T may be inherently faster than those of L. The drawbacks of this technique are that it is essentially a manual operation, and that correctness of *genf* with respect to f is difficult to prove.

- **Partial Evaluation** The idea of partial evaluation is to automatically specialize the code of f to the supplied input, in a manner similar to standard evaluation. In particular, this means defining a program in language C called *pe*, which we call a partial evaluator for language L. This means that we can pass to *pe* the implementation of a function f along with f's immediate input x, and get back a version of f that is *specialized* to x. This specialized version, also called a *residual*, is often denoted as f_x. Equationally, we have

  $$[pe]_C\ (f, x) = f_x$$
  $$[f_x]_L\ y = [f]_L\ (x, y)$$

  for all f, x, and y. As seminally observed by [Futamura 71], a partial evaluator and interpretter can be used together to achieve the same result as manual code generation, automatically! Using the `fftw` example, we see that `dft` $=$ *pe* $($`fft`$, x)$, where x is the dimension of the dft. This has the benefit of being an automatic process, although the generated compiler is only as good as the partial evaluator. Much of the subsequent work in partial evaluators has been with the purpose of chasing this goal.

- **Stage Separation** The final technique, stage separation, is conceptually the simplest. The idea is to define in C a program called ps, which is a path separator for language L. It works by cleaving the L-function f into two L-functions f_1 and f_2, where f_1 builds a data structure from the available input, and f_2 consumes that data structure as well as the delayed input to produce the standard ouput. Equationally, this is

  $$[pe]_C \ f = (f_1, f_2)$$
  $$[f_2]_L \ ([f_1]_L \ x, y) = [f]_L \ (x, y)$$

  for all f, x, and y. This technique, performed manually, is a common exercise for every programmer. In can be considered the general form of many common compiler optimizations such as loop hoisting, an example we will consider later. The key difference between stage separation and partial evaluation is that the former is performed in absence of the first input. This suggests that stage seperation is applicable in more scenarios than partial evaluation, at the cost of being less aggressive. Also, whereas stage seperation creates a data structure to represent the intermediate progress of a computation, partial evaluation usually embeds that datastructure in the code of the residual [some related work says it can alleviate this]. For these reasons, stage splitting should be thought of as a mechanism for *algorithmic* optimization, rather than *systematic* optimization.

In this work we present a stage separation algorithm for a fragment of ML. Specifically, our formulation can separate terms containing sums and recursion, which have not appeared previously in the stage seperation literature. We anticipate that the algorithm will also be able to also separate this language extended with with first-class functions, although this is left as future work. Additionally, our presentation is more explicitly type-motivated than those that have come before.

2 Binding Time Analysis

Previous research has observed that the stage separation problem, as defined above, is inherently ambiguous (indeed, this is true of all the staging techniques). That is, there can exist multiple ways to partition f into f_1 and f_2 that satisfy $f_2(f_1(x), y) = f(x, y)$. For a trivial example, we can set f_1 to be the identity and f_2 to be f. We can resolve this ambiguity by requiring that f be given to us in a form which unambiguously specifies the part of the computation to be considered first-stage, and the part of the computation to be considered second-stage. The particular form that we employ, introduced later, is suggested by [Pfenning and Davies 2001], which explores the connection between staging phenomena and modal type systems. That work also shows the equivalence of the form we use and others, such as the two-level languages of [nielson^2].
The process of adding staging annotations to an otherwise ambiguous computation is called *binding time analysis*. We do not explore binding time analysis in this work, and instead assume that complete annotations are given by the programmer or by some analyzer.

3 λ^{12}

(Introduce the language, talk about its type system and maybe its evaluation.)

4 Examples

Some examples to get us comfortable with the language.

4.1 loop hoisting

5 Stage Seperaion Algorithm

Talk about the goal of the transformation. Introduce the general form with open variables and a context, which is *not* the simplistic f_1, f_2 thing. Stress the first image/second image/boundary type stuff.

5.1 Type translation

5.2 Term translation

State the main theorems we want to be true. Refer to the

5.3 Functions

Talk about why the current set up for functions alleviates the boundary-type guessing problem.

5.4 Issue with sums

Sums, as done, are bad.

6 Implementation

Not much to discuss. It exsists. Talk about how crufty the naive transformation is.

7 Related Work in Stage Seperation

Stage seperation was introduced in [Jorring and Scherlis 86], then called "pass seperation." They gave an algorithm that simultaneously performed a binding time analysis and stage seperation for a recursive first-order language with if statements and primitive operations (essentially, the core of Java).
[Hannan 94] gives an algorithm for simultaneous binding time analysis and stage seperation for a class of languages called "abstract machines".
[Knoblock and Ruf 96] does BTA and PS for non-recursive imperative shader programs. They pay a lot of attention to the size of the intermediate data structure, since memory is at a premium in their shaders.