General Stage Untangling with Applications to Algorithm Derivation

Nicolas Feltman et al.

December 11, 2013

1 Introduction

Often when programming high-performance systems, a programmer will wish to specialize a bivariate function $f : \alpha \times \beta \rightarrow \gamma$ to one of its inputs, $x : \alpha$, while leaving the other, $y : \beta$, for later. This need may arise because x varies at a higher frequency than y, and we would wish to avoid repeated calculation. It may also be the case that x is available at a time when computational resources are less expensive than when y becomes available. It's even possible that knowledge of x allows further code optimizations that would have otherwise been unwise to speculate.

Consider two examples: the discrete Fourier transform (DFT) and a raytracer. In the DFT case we say that $x : \mathbb{N}$ is the dimension of the transformation, $y : \mathbb{R}^x$ is the data to be transformed, and f simply computes the DFT. Much of the structure of modern DFT algorithms depends on x (for both mathematical decomposition and cache performance reasons) but not on y, so a great deal of work could be done to specialize the algorithm to a particular number of dimensions. In the raytracer case, we say that $x : \texttt{Geom}$ is a specification of the scene geometry (perhaps a list of triangles), $y : \texttt{Ray}$ is a ray cast from some point in the scene, and f computes the first intersection of the ray and the geometry. In modern raytracers, we may cast millions of rays without varying the scene geometry, so it could be advantageous to precompute as much of the algorithm as possible based on only scene geometry.

We say that the problems above exhibit a *staged* structure, in that they can be broken down into parts that run at different points in time (*i.e.* stages). There are a variety of techniques to take advantage of this staged structure. The general theme is to do some work with the immedaite input (x), producing an intermediate result, and then finish the rest of the computation once the delayed input (y) is available. We consider three techniques: program generation, partial evaluation, and pass separation. Each technique has broader applicability than the previous, but is less aggressive of an optimization.

1.1 Program Generation

A common solution is to write code that writes code. Essentially, second-stage computations are represented by second-stage code, and second-stage code is a value that can be passed around and manipulated by first-stage code. To use the terminology from above: this first stage code is a function that takes in x and outputs second stage function, which itself takes in y and outputs the final answer. Note that this is essentially a manual process, in that the programmer must know the function f which is being specialized. Using the equational notation of [Jones 96], where "$[\![h]\!]\ z$" means the "the code h interpreted as a function and applied to input z," we say that a programmer defines a program generator g that satisfies

$$[\![[\![g]\!]\ x]\!]\ y = [\![f]\!]\ (x, y)$$

for all x and y. When well supported by the language (see terra, metaml) this technique is often called *metaprogramming*. Of course, even without first class support, one can represent the second stage code using strings or other data types. Defined so broadly, program generation includes all compilers, where f is a hypothetical interpretter, g is the actual compiler, x is a program in the source language, and y is the input to x. Program generation has also been used to implement the DFT example in the form of FFTW [cite this].

The benefit of the program generation technique is full control, even up to the ability to use domain-specific optimizations. This comes at the cost of essentially being a manual operation.

1.2 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 pe, called a partial evaluator, to which we can pass 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

$$\begin{aligned} [\![pe]\!]\ (f, x) &= f_x \\ [\![f_x]\!]\ y &= [\![f]\!]\ (x, y) \end{aligned}$$

for all f, x, and y. As seminally observed by [Futamura 71], partially evaluating an interpretter on a source program is equivalent to compiling that program to whatever language the interpretter was written in! This has the benefit of being an automatic process, although the effective 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.

Partial evaluation is a more general technique than program generation, in that the partial evaluator is defined once and works for all fs that need to be specialized. Of course, the partial evaluator has no domain knowledge of the program it is splitting, so it cannot be nearly as aggressive as a program generator might be. The two techniques are similar in that they both require the first input x to do their specializing action.

1.3 Pass Separation

The final technique, pass separation, is conceptually the simplest. The idea is to define a program called ps, for *pass separator*. It works by cleaving the function $f : \alpha \times \beta \to \gamma$ into two functions $f_1 : \alpha \to \tau$ and $f_2 : \beta \times \tau \to \gamma$ for some type τ, where f_1 builds a data structure (of type τ) from x, and f_2 consumes that data structure as well as y to produce the standard ouput. In analogy to partial evaluation, we call f_2 the *residual*. Equationally, this is

$$\begin{aligned} [\![pe]\!]\ f &= (f_1, f_2) \\ [\![f_2]\!]\ ([\![f_1]\!]\ x, y) &= [\![f]\!]\ (x, y) \end{aligned}$$

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 common compiler optimizations such as loop hoisting, an example considered later. Like partial evaluation, pass separation is an automatic operation, but unlike both partial evaluation and program generation, pass separation does not require the value of the first argument, x, to properly specialize f. This makes pass separation the most widely applicable of the staging techniques, at the cost of being able to perform aggressive optimizations in the residual that might depend on the first argument.

In this work we present a pass 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 and Stage Untangling

The pass separation problem can naturally be decomposed into two subproblems:

- Decide which parts of the definition of f belong to which stage and produce an annotated version of f.
- Use the annotated version of f to produce f_1 and f_2.

An analogous form of this decomposition exists in the partial evaluation literature, wherein the first subproblem is known as *binding-time analysis.* The second subproblem, in which the difference between pass separation and partial evaluation is more manifest, does not seem to yet have a name. We call it *stage untangling.* Unlike previous work in pass separation, we focuses entirely on the stage untangling problem by assuming that f is given to us in an annotated form.

Previous research has observed that the pass separation problem is inherently ambiguous in that the equations defined above do not fully specify the definition of *pe*. 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. Fortunately, this ambiguity is contained entirely within the binding-time analysis portion of pass separation; stage untangling is entirely determined.

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

Pass seperation was introduced in [Jorring and Scherlis 86]. They motivating pass separation as a technique using hand-separated examples, butstopped short of providing an algorithm to do this automatically.

[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.

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".