Stage-Splitting a Modal Language

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Staged computation has existed as a programming technique for over three decades. Jorring et al. ([5]) identify three classes of staging techniques: meta-programming, partial evaluation, and stage-splitting. The first two of these have received significantly more attention than the third.

We call these techniques *staged* in that part of their input comes in the first stage, and part comes in the second stage. Likewise, some computation occurs at each stage as well.

Countless meta-programming systems exist (...twelve thousand citations...[3]), and their background theory and type-systems are well understood ([2]). Partial evaluation, too, is well-understood. Partial evaluation systems exist... . This paper explores both theory and applications for stage-splitting.

2. Stage-Splitting Definition and Comparison to Partial Evaluation

First, we review the definition of partial evaluation. Informally, a partial evaluator takes the code for a function f, as well as the first-stage input x to that function, and produces the code for a version of that function *specialized* to the first input, often called f_x. This f_x function can then be evaluated with the second-stage input to produce the same final answer that f would have. The goal of the process is that f_x should be cheaper to evaluate than f, although this can't be guaranteed for all inputs. We now state this theorem more formally: a partial evaluator is some function p such that,

$$\forall f, x. \exists f_x. [p(f, x) = f_x \text{ and } \forall y. [\![f]\!](x, y) = [\![f_x]\!](y)]$$

where (borrowing notation from [4]), $[\![f]\!]$ means the mathematical function corresponding to the code given by f.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Informally, we define stage-splitting to be the process of taking some function f into two other functions, f_1 and f_2, where f_1 computes a partial result from the first-stage input, and f_2 uses that partial result and the second-stage input to compute a final result which is the same as if we had just run the original f on both inputs. Again, more formally, a stage-splitter is some s such that,

$$\forall f. \exists f_1, f_2. [s(f) = (f_1, f_2) \text{ and } \forall x, y. [\![f]\!](x, y) = [\![f_2]\!]([\![f_1]\!](x), y)]$$

We first discuss a few similarities between partial-evaluation and stage-splitting. First off, both techniques have the same form of input, namely a bivariate function where the first input comes at stage one, and the second input comes at stage two.

Again in both cases, the governing equations are too weak to fully determine the definitions of p and s. Indeed, both admit completely trivial definitions. Consider the stage-splitter which always returns the identity for f_1 and f for f_2, or analogously the partial evaluator which always returns an f_x that just closes over the input x and internally calls f once y is available. The ambiguity of these equations (modulo standard program equivalence of the outputs) can be resolved by adding annotations to f to clearly specify the parts of the computation that are first stage and the parts that are second stage. Later, we show that the same annotations suffice for both partial evaluation and stage-splitting.

The differences between stage-splitting and partial evaluation are likewise evident from these governing equations. For instance in partial evaluation, the existential f_x depends on x, which means that the partial evaluator cannot be run until x is known. Moreover, if one wishes to specialize f for multiple x's, then the partial evaluator must be run several times. Depending on the use case and cost of partial evaluation, this may be prohibitively expensive. Alternatively, a stage-splitter need only be run once, and this can be done entirely before any x is known.

We can recover a valid partial evaluator from a stage-splitter by stage-splitting the input function f into f_1 and f_2, computing $[\![f_1]\!](x)$ to obtain $\bar{x}$, and then returning an f_x such that $[\![f_x]\!](y) = [\![f_2]\!](\bar{x}, y)$. However, one *cannot* construct a valid stage-splitter out of a partial evaluator, since a partial evaluator by definition depends on the first-stage input, whereas a stage-splitter does not have that available.

Note that this does not mean that stage-splitting is a strict generalization of partial evaluation. In practice, partial evaluators easily perform optimizations (such as branch elimination, discussed later) which are beyond the scope of stage-splitting, and would require further technology than has been developed here. It is best to think of stage-splitting as simply the first half of partial evaluation, where the back half is an optimizer. [Might be able to come up with a futamura projection-like statement here, which would be really really really cool.]

3. A Two-Stage Modal Language

Introduce the next and prev concepts, along with typesystem. Introduce binding time analysis here, and explain that we don't care about it. Show some examples. Introduce a hold operation.

4. Straw Semantics

Introduce the straw semantics, previously known as the "erasure" semantics.

5. Data and Control Dependency

State a data-dependency theorem (that stage 1 values cannot depend on stage 2 values). It should be true for the straw semantics! Also state a control dependency theorem (that the execution of stage 1 code can't depend on stage 2 values). This won't be true for the straw semantics, and show counterexample.

6. Diagonal Semantics

Present another semantics which obeys non-backwards control dependency. This might be called the diagonal semantics. Go through an example such as currying to build intuition for why this is really necessary (i.e., if we don't have these weird rules, currying breaks down!).

7. Splitting Algorithm

Present the splitting judgement. Give statements of type and value correctness for splitting. Give all of the splitting rules. Talk through a few of them.

8. Examples for Staged Pipelines

Give the gist of one-to-one pipeline example (like client/server). Then talk about a one-to-many pipeline. Then talk about a many-to-one pipeline like spark. It clear how to target something like this for known base types on the boundary, and for product types. But sums on the boundary are hard! We leave many-to-one as future-work.

9. Examples of Algorithm Derivation

Fast exponent example. Quickselect example.

Things to try: an interpreter which, partially evaluated, does cps or something.

For each of these examples, talk about what partial evaluation would do and why that might be bad.

10. Related Work

Our stage-splitting algorithm was first suggested in [5] under the name *pass separation*. They essentially proposed that a function f could be split into two others, f_1 and f_2, such that $f(x, y) = f_2(f_1(x), y)$. They did not distinguish binding time analysis from stage splitting, and so pass separation inherits the former's ambiguity. The main goal of [5] was to motivate pass separation and other staging transformations as a powerful way to think about compilation and optimization. Accordingly, their approach was entirely informal, with no implementation realized. Moreover, they predicted that "the [pass separation] approach will elude full automation for some time."

Implementations of the stage-splitting algorithm have appeared in the literature exclusively (and coincidentally) in the context of graphics pipelines. The first of these ([6]) uses a binding time analysis to separate those parts of graphics shaders that are input-invariant from those which are not, and then uses a stage splitting algorithm to factor that into two shaders, thereby minimizing recomputation. Their shaders are written in a C-like language with basic arithmetic and if statements. Although their analysis does not give an explicit account of the type-level behavior of the splitting algorithm, it effectively can synthesize product and sum boundary type.

Like the previous example, the Spark language ([**?**]) uses staging to minimize recomputation in real-time rendering applications. But instead of using a binding-time analysis, Spark allows the programmer to manually target stages of the graphics pipeline. Since the modern graphics pipeline is inherently a many-to-one system, this is difficult to reconcile with sum types on the boundary. Fortunately, Spark has a set of syntactic restrictions that prevent sum boundary types. Spark does not clearly identify this conflict, but the authors did note that first-stage if statements were difficult to provide meaning to [need quote].

[RTSL and SH]

[Discuss Yong's recent paper here. It does some pretty sophisticated binding time analysis, with a somewhat straightforward splitting after that. They have the same many-to-one use case as Spark, but syntactic restrictions prevent sum types on the boundary, sort of. If we wanted to faithfully represent their system in ours, we would need some mechanisms for abstraction over stage, which we do not have.]

Davies ([1]) explored the connection between linear temporal logic and its corresponding type system [circle](which we adapted into [circle sub 2]), and showed the equivalence between [circle] and existing systems for binding time analysis. That work provided β and η rules for the next and prev operators, but did not consider a full dynamic semantics for the whole language. Whereas [name of our type system] is appropriate for stage-splitting and partial evaluation, [2] provides a similar system, [square], that is appropriate for meta-programming. The main difference is that terms inside a [prev] operator do not see any stage-2 bindings declared outside of it. They note that where [circle] corresponds to the non-branching temporal logic, [square] corresponds to the branching version.

[Meta-ML eases off on this restriction but does not (I think?) eliminate it.]

[What's going on with names and necessity?]

[Our work bears a lot of similarity to ML5, which also uses a modal type system. The difference is that we target stages systems (each stage talks to the next), whereas they target distributed ones (all stages talk to all others). The type systems reflect this directly in the world accessibility relation. There might be some analogue of stage-splitting in the ML5 work, but I have not yet isolated it (might be buried in CPS conversion).]

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[1] R. Davies. A temporal-logic approach to binding-time analysis. In *LICS*, pages 184–195, 1996.

[2] R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.

[3] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. . URL `http://doi.acm.org/10.1145/2491956.2462166`.

[4] N. D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, Sept. 1996. ISSN 0360-0300. . URL http://doi.acm.org/10.1145/243439.243447.

[5] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 86–96, New York, NY, USA, 1986. ACM. . URL http://doi.acm.org/10.1145/512644.512652.

[6] T. B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 215–225, New York, NY, USA, 1996. ACM. ISBN 0-89791-795-2. . URL http://doi.acm.org/10.1145/231379.231428.