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

Introduction

2. A Two-Stage Modal Language

Introduce the next and prev concepts, along with typesystem. Show some examples. Introduce a hold operation.

3. Straw Semantics

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

4. 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.

5. Diagonal Semantics

Present another semantics which obeys non-backwards control dependency. This might be called the diagonal semantics.

6. 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.

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

7. 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.

8. Examples of Algorithm Derivation

Fast exponent example. Quickselect example.

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

9. Related Work

Our stage-splitting algorithm was first suggested in [**?**] 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 nondeterminism. The main goal of [**?**] 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 ([**?**]) 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 re-computation. 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.

[Discuss Yong's recent paper here. It does some pretty sophisticated binding time analysis, with a somewhat straightforward splitting after that. The difference between this work and everything mention prior, is that first stage results aren't exactly . 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.]

In addition to introducing pass separation, [**?**] also distinguishes it from two other staging techniques: partial evaluation and metaprogramming.

[In brief, the three approaches can be distinguished based on when the first stage evaluation occurs, relative to when the compiler takes action. In our work, the splitting occurs entirely before the first stage values are known. In partial evaluation, these things happen essentially simultaneously. For this reason, partial evaluation can do things like branch elimination for first-stage cases, which we cannot do.]

[Our type system is adapted directly from [**?**]. It was originally developed to describe binding time analysis for use in partial evaluation.]

[Metaprogramming is more different still. It involves the creation of values that correspond to second stage code, often also with a primitive to allow the evaluation of second stage code to produce a first stage result. For a type system that does that, you want [**?**]. Notice the closed-code restriction. 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] P. Q. Smith, and X. Y. Jones. ...reference text...